Titre: Title:	Implementation and Evaluation of Counting-Based Search for Table Constraints in the OscaR Solver
Auteur: Author:	Jinling Xing
Date:	2020
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Xing, J. (2020). Implementation and Evaluation of Counting-Based Search for Table Constraints in the OscaR Solver [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/4196/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/4196/
Directeurs de recherche: Advisors:	Gilles Pesant
Programme: Program:	Génie informatique

Implementation and Evaluation of Counting-Based Search for Table Constraints in the OscaR Solver

JINLING XING
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Janvier 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Implementation and Evaluation of Counting-Based Search for Table Constraints in the OscaR Solver

présenté par **Jinling XING**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Guillaume-Alexandre BILODEAU, président
Gilles PESANT, membre et directeur de recherche
Michel GENDREAU, membre

DEDICATION

Thanks to my parents,
for supporting me to receive the best education possible...

Merci à mes parents,
de m'avoir soutenue pour recevoir la meilleure éducation possible...

ACKNOWLEDGEMENTS

Thanks to my director Gilles Pesant, helping me during my studies patiently and he was the guiding light for me to complete this master thesis. I would like to thank Luc Michéa, encouraging me to believe that I can keep improving during my studies. For the members of Quosséça lab, I would also like to thank their generous help.

RÉSUMÉ

Dans ce mémoire, nous allons nous intéresser à *la programmation par contraintes*, un outil efficace en ce qui concerne la résolution de problèmes combinatoires. Nous allons nous intéresser aux problèmes utilisant les contraintes *table* et plus particulièrement leur implémentation compacte qui a été grandement améliorée par l'utilisation de la structure de données «sparse bit set» réversible. Nous contribuerons en créant une heuristique de recherche basée sur le *dénombrement* utilisant l'information sur les supports des contraintes *table*.
Nous allons implémenter puis évaluer un algorithme de dénombrement sur *Oscar*, une librairie de résolution de problèmes par contraintes créée pour résoudre les problèmes combinatoires. Nous définirons alors un algorithme pour obtenir les supports et un algorithme pour réaliser une heuristique de recherche utilisant les informations précédentes. Tous ces algorithmes ont un but commun, mettre en place une recherche basée sur le dénombrement.
Nous allons expliquer les modifications faites à *Oscar* et les heuristiques de recherche que nous avons créées dans *Oscar*. Finalement, nous présenterons nos résultats sur différents exemplaires de problèmes et nous analyserons les résultats comparés à l'état de l'art pour en déduire les améliorations apportées. Nous utiliserons pour cela les algorithmes suivants: *dom* et *dom/deg*. L'expérience montrera que notre recherche basée sur le dénombrement est compétitive pour les exemplaires complexes mais qu'elle coûte plus de temps dans certains cas, même si nous avons moins d'échecs.

ABSTRACT

In this thesis, we work on *constraint programming*, an efficient approach to solve combinatorial problems. We consider problems using table constraints and in particular the compact table implementation. Reversible sparse bit sets have been used for the compact table implementation recently, and it improves its efficiency. We contribute by making the heuristic search more efficient for such problems by using counting-based search. Counting-based search uses the supports information from the reversible sparse bit set data structure (used to maintain supports in the table constraints).

We implement and evaluate our contribution in *Oscar*, a constraint programming solver to solve combinatorial problems. We explain the modifications we made in *Oscar* and the heuristic searches we created in *Oscar*. We define an algorithm to get the supports from table constraints, a variable ordering heuristic search, and a value ordering heuristic search. All of these algorithms work toward the same goal, counting-based search. Finally, we present our results on different instances of problems and analyze the results and improvements. We compare our methods with *dom* and *dom/deg*. The experiment shows counting-based search is competitive if the instances are hard and it costs more time in some instances even if we have fewer failures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

CP	Constraint Programming
CT	Compact Table
CSP	Constraint Satisfaction Problem
CSPs	Constraint Satisfaction Problems
Oscar	Operation Research in Scala
ABS	Activity Based Search
IBS	Impact Based Search
CBS	Counting Based Search
XML	eXtensible Markup Language
UML	Unified Modeling Language
AC	Arc Consistency
WDEG	Weighted Degree Search
DOM/WDEG	Domain/Weighted Degree Search
SVOs	Static Variable Ordering Heuristics
DVOs	Dynamic Variable Ordering Heuristics
DFS	Depth-first Search
GCC	Global Cardinality Constraint
GAC	Generalized Arc Consistency

CHAPTER 1 INTRODUCTION

In industry, many planning and scheduling problems can be framed as combinatorial problems. Constraint Programming (CP) is one computational approach to solve them. CP expresses problems declaratively through a formal mathematical model. Users can describe their problems without having to specify how they should be solved. In Oscar [1], we can use CP to solve combinatorial problems. There are many constraints that can help us to form our CP models for specific problems. In this thesis, we work on table constraints because the table constraint generalizes many other constraints and is widely used in many problems. Compact table [2] is an efficient implementation of the table constraint that uses the reversible sparse bit set techniques. When users create their models with constraints, they need to give the solver which heuristic search they want to use to search a solution for their problems. The search strategies decide how to explore the search space and build the search tree; users can use different search methods to solve their problems; different methods have an impact on the search tree and the search space. In this thesis, we consider Counting Based Search (CBS). The required algorithms have been designed for several constraints [3] [4] [5] but not for the table constraint yet. In this thesis, we want to check if CBS improves the efficiency of solving CP models featuring table constraints.

1.1 Basic Concepts

CP is a method to solve Constraint Satisfaction Problems (CSPs). A Constraint Satisfaction Problem (CSP) is described by variables, variables' domains, and constraints restricting the variables. When we want to find a solution of a CSP, propagation and backtrack might occur. The search method will decide how to explore the search space and affect how fast we can find a solution. The related concepts of CSPs and CP will be introduced in this chapter.

Definition 1.1.1 (Constraint)

A constraint is a *relation* over a subset of the variables, called its *scope*. The *arity* of the constraint is the size of the scope.

The *arity* of constraints includes *unary* constraints that concern only one variable, *binary* constraints that link two variables and *higher-order* constraints that associates three or more variables.

The *Constraint graph* uses nodes and arcs to represent the variables and constraints. The constraint graph shows the relationship between variables and constraints for CSPs. The 4-Queens problem is a simple example from n-Queens. It aims at placing four queens on a 4×4 chessboard in which none of them can attack each other directly. The nodes represent the queens whereas arcs represent the constraint that no two queens are in the same row, same column, and same diagonal in the 4×4 chessboard because queens can attack in their rows, columns, and diagonals. Figure 1.1 shows the constraint graph of the 4-Queens problem.

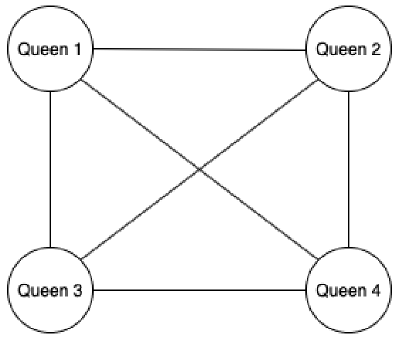

Figure 1.1 Constraint graph for 4-queens

Definition 1.1.2 (CSP)

A CSP can be represented by a triple $< X, D, C >$ where X represents a finite set of variables $\{X_1, X_2, ..., X_n\}$, D is a set of domains associated to each variable $\{D_1, D_2, ..., D_n\}$, C is a set of constraints $\{C_1, C_2, ..., C_m\}$ that will restrict the variables' domains.

The constraint *Alldifferent*$(x_1, ..., x_n)$ [6] states that all the pairwise values are different over a set of variables. The *table* constraint is also called extensional constraint, and it is widely used in Constraint Satisfaction Problems (CSPs) and is adaptable with other constraints. The table constraint can express the relation of CSPs straightforwardly, and it is applied in many CSPs with other constraints, such as scheduling problems that used $Alldifferent$ and *table* constraints. Table constraint is simple to represent the relations and easy to adapt with other constraints such as $Alldifferent$. For example, we have variables x_1, x_2, x_3 and the corresponding domains $D_1 = (2, 9)$, $D_2 = (2, 3)$, $D_3 = (1, 5)$, so the set of tuples of all the variables that restricted to table constraint are $\{(2, 2, 1), (2, 2, 5), (2, 3, 1), (2, 3, 5), (9, 2, 1), (9, 2, 5),$ $(9, 3, 1), (9, 3, 5)\}$, and the set of tuples of all the variables restricted to table constraint and $Alldifferent$ constraint are $\{(2, 3, 1), (2, 3, 5), (9, 2, 1), (9, 2, 5),$ $(9, 3, 1), (9, 3, 5)\}$.

For the 4-Queens problem, variables are our Queens, each of them will be distinguished by the index i in X_i, X_i represents in which row the queen of column i is placed. D is the domain of each variable, D_i is the domain of X_i, $D_i = \{1, 2, 3, 4\}$. In the constraints, C_1 is $AllDifferent(X_i)$ that means the rows are different. C_2 and C_3 are $AllDifferent(X_i + i)$ and $AllDifferent(X_i - i)$ that describes the diagonals are distinct.

The 4-Queens problem will be used to explain the constraint propagation. The cross marks on the chessboard represent the places where we cannot place a queen anymore. The red crosses on the diagonal tell no queens can be on the same diagonal and the blue crosses on the row mean that no two queens can be on the same row. The process of eliminating certain squares is called *propagation*. Figure 1.2 shows the constraint propagation we mentioned above.

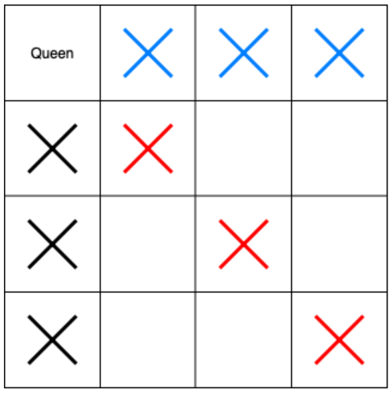

Figure 1.2 Constraint propagation for 4-queens

Definition 1.1.3 (support) A value v_i in the domain D_i is supported in the scope of a constraint c if there is a valid tuple in the corresponding relation where $x_i = v_i$.

Definition 1.1.4 (solution) For a CSP, if there is an assignment of a value v_i to each variable x_i that satisfies all the constraints c_j, the assignment $x_i = v_i$ (i in $\{1, 2, ..., n\}$) is a solution for that CSP.

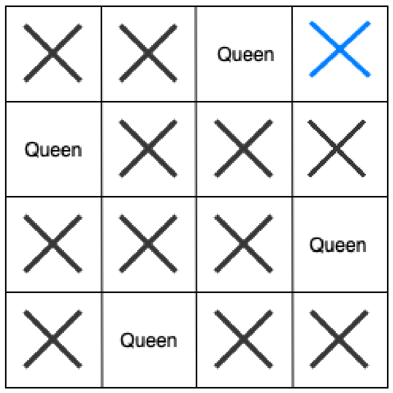

Figure 1.3 One of the solutions for 4-Queens

The 4-Queens problem has 256 possible configurations. Figure 1.3 shows one of the two solutions to the 4-Queens problem.

If the constraint propagation cannot filter more values from the domain of variables, the search will occur to explore the solution space. In this thesis, the search is based on the supports for each variable-value pair. The next section will use an example to explain the CSPs and the detailed process of the search.

1.2 Search for CSP

The core of CP is filtering and search, filtering removes values from non-feasible solutions, search gives you rules to explore the solution space. We want to solve a CSP problem, but failures can occur during the search process. When a failure occurs, the algorithm has to *backtrack*. A backtracking search will explore the search space according to the Depth-first Search (DFS) strategy. When the node explored is part of the solution, we continue the search from this node. Otherwise, backtracking will occur. Constraint propagation, binary branching strategies, and the variable-value heuristic ordering method are the techniques to improve the performance of the backtracking algorithm. In this thesis, we wish to show that CBS with the compact table will be a fast combination to find solutions. In Figure 1.4, the blue cross shows the failure occurred, and the third column cannot place a queen that satisfies the constraints, so the algorithm will backtrack.

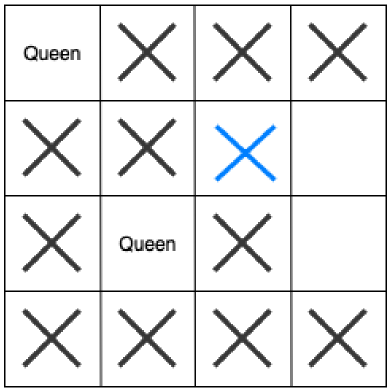

Figure 1.4 Backtrack for 4-Queens

Constraint propagation filters the locally inconsistent values, but it is not sufficient to find a solution. The search algorithm will help the solver to make a decision. Sometimes we arrive at a point where we removed all the values we could, but we did not solve because some variables still have several values available. We then have a choice to make between those values available, hence a search.

Which variable will be the first one to branch on, and which value should be assigned to the variable? Most of the existing search heuristics use data at the individual variable level such as the domain of variables. The search algorithm based on the individual information needs to set the rules to break ties sometimes. For example, if we choose to search on the smallest domain first, this search algorithm would not work well when a lot of variables have the same domain size.

Here, we will use the 4-Queens problem as an example to illustrate the binary branching; this example uses three different constraints (No queens on the same row and no queens on the same diagonal). This example shows how the CSP is defined and how to find a solution using the lexicographic order of the variable. That makes the explanation of basic concepts of constraint programming clearer.

To solve this problem, we need to mention that the propagation of constraints will eliminate inconsistent values from the domain, and search strategies will be used to find variable-value assignments. Here is a search algorithm we created for the 4-Queens problems. The search method will choose the unbound variable first according to the ascending order, and the smallest value will be assigned to the variable. Following the search approach we created, we set the variable $X_1 = 1$; then the constraint propagation will remove the values of other variables' domain, Figure 1.2 shows the result. Here we need to specify that $X_2 = 3$ means the queen will place in the second column and the third row. The variable X_2 can only choose the third or the fourth position. Figure 1.4 shows the result after the propagation. In the third column, no queen can be placed any more, and failure occurs.

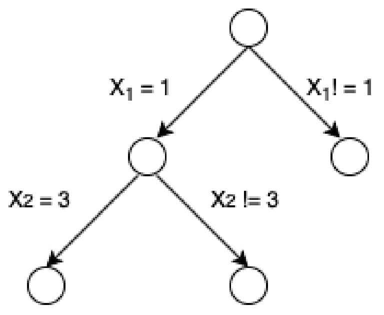

Figure 1.5 Partial search tree for 4-Queens

Binary branching has up to two branches per node, the left branch, and the right branch. The search algorithm needs to tell the current node how to generate the child nodes. The 4-Queens example shows that the left branch corresponds to a variable being assigned to a chosen value and the sub-tree of this branch will be explored until a failure occurs; the right branch is the other situation. Like the partial search tree of 4-Queens, the left branch is $X_i = V_i$ and the right branch is $X_i! = V_i$.

1.3 Research Objectives

Table constraints are very common in CP models. CBS has already been implemented in other solvers. In Gecode [7], Gagnon and Pesant [8] presented several works to accelerate CBS. Their work experiments have shown that using CBS can achieve orders of magnitude faster if the benchmark instances are hard. To use CBS on models with table constraints we need to count supports efficiently. The reversible sparse bitset data structure (see Chapter 2) will help us to get the solution density faster. If we get the support count from the compact table in constant time, the solution density based on support count will not be hard. Second, the compact table constraint is efficient in many cases and it is applied in many problems, so if CBS for the compact table constraint is better than other searches, many real problems will be solved faster.

In this thesis, we focus on CBS that uses *solution density*. First, we need to understand the existing classes and packages in the *Oscar* solver and the architecture of *Oscar* and assure the classes and packages are related to our research. Second, we need to focus on the table constraints and the corresponding data structure to make sure it is a good start of the modifications in the constraint side. Third, we need to consider all the issues when we implement the *counting-based search* for table constraints in the *Oscar* solver. Last but not least, we want to optimize the *counting-based search* for table constraints in *Oscar* to solve real problems.

1.4 Thesis Outline

Chapter 1 was the introduction. Chapter 2 is the literature review on the table constraint and heuristic search in the CP field. Chapter 2 also explains the methods related to the compact table and the reversible sparse bit set data structure. Chapter 3 explains how to calculate the sum of supports and how to store the sum of supports for the CBS implementation. We will introduce the Oscar software architecture in Chapter 3 and the changes we implemented. Chapter 4 will show the experimental results as well as the analysis of the performance on the different benchmark instances. The last part will be the conclusion of this thesis and discuss further research.

CHAPTER 2 RELATED WORK AND THE FUNDAMENTALS OF CONSTRAINT PROGRAMMING

This chapter has two parts. Section 2.1 presents the fail-first principle and the literature review of the main heuristic searches in constraint programming. A review of the main branching heuristics in constraint programming is presented, in particular, the CBS based on the solution density. Section 2.2 describes the principle of the Compact Table (CT) constraint and the methods related to this constraint. The reversible sparse bit set is the data structure of supports, and we introduce the principles of the reversible sparse bit set and use an example to explain the details.

2.1 Survey Related to Heuristic Search

A good heuristic search in constraint programming explores few nodes to find a solution or to prove that no solution can be found. A CSP can be solved using a backtrack search. Which variable to branch on and which value to assign to it? Variable ordering and value ordering are the keys to make the decisions during the search. Recently, the variable ordering gets more attention from researchers than value ordering. There are two families of variable orderings: Static Variable Ordering Heuristics (SVOs) and Dynamic Variable Ordering Heuristics (DVOs). SVOs fix the order in which variables will be branched on before search. DVOs consider the information at the current search tree node and use it to guide the search.

2.1.1 Fail-First Principle

An efficient tree search method for CSPs created by Haralick and Elliott [9] is known as the *fail-first principle*. The fail-first method explores the search space where the failure will occur first, hoping that the search tree will be reduced and the search space will be minimized. For example, if the method chooses the variable with the minimum domain size first which means it has the fewest values to assign to this variable, the information will be gathered in the initialization step and also in the search processing, then the method will choose the node with the fewest values in its domain. The sub-tree of the variable that has fewest values in its domain will have fewest branches. In this case, the fewest nodes will be explored when the method searched the variable that has the minimum domain size and the depth of the search tree will be minimized and the search space will be minimized too. Smallest-domain-first is the usual implementation of this principle.

2.1.2 Conflict-Driven Heuristic

The traditional dynamic variable search considers the current nodes' status and the other information like the degree of variables (i.e., in how many constraints they appear) and the domain size of variables. Boussemart et al. [10] proposed a method using the previous information of nodes to guide the search, and the new dynamic variable search is denoted Weighted Degree Search (WDEG). Place a counter for each constraint to save the information during the search, and the hardest constraint will occur in most conflicts, then this constraint will be considered as the most important constraint and assigned the biggest weight. WDEG follows the fail-first principle: it will choose the variables related with the most weighted constraints and search the hard part of the problem at the beginning. The information saved during the search is the weight for each constraint, when the domain of a variable is empty, the counter of weight for the constraint involved in the problem will be increased during the search processing part. Choosing the variable with the smallest ratio of current domain size to the current weighted degree is a variant named Domain/Weighted Degree Search (DOM/WDEG) [10]. DOM/WDEG is considered a state-of-the-art generic heuristic and is implemented in most CP solvers.

2.1.3 Impact-Based Search

Impact Based Search (IBS) [11] is motivated by a criterion named *pseudo-cost*. By rounding up or down the value of the variable, we derive an estimate of the change in the objective function per unit, called the *pseudo-cost* of the variable. IBS is based on the concept of *impact*. The *impact* is the element to describe the importance of the variable-value assignment in the reduction of the search space. IBS chooses the greatest impact on variables and chooses the smallest impact on values. This principle is also like most of the methods that choose the hardest part to explore first to reduce the search space, such as choosing the minimum domain size first. The *impact* of a variable is the sum of the impact of the values in its domain. To understand the principle more, there are some formulas to describe the impact and the relevant concept. Generally, the number of variable-value pairs can be described using *the Cartesian product*:

$$P = |D_1| \times |D_2| \times \cdots \times |D_i| \times \cdots \times |D_n|$$

When the value is assigned to the variable, the *Cartesian product* before the assignment and after the assignment will change. If this assignment can reduce the search space efficiently, the P_{after} will be smaller, the ratio of P_{after} and P_{before} will be smaller too, the impact of this variable-value assignment will be close to value 1.

$$I(x_i = a) = 1 - \frac{P_{after}}{P_{before}}$$

2.1.4 Activity-Based Search

Activity Based Search (ABS) [12] collects the constraint propagation information to measure the activity of each variable. ABS needs a counter for each variable to save information about activities. The activity of a variable is the frequency at which the domain of that variable is reduced during the search. ABS will first choose the variable with the highest activity.

2.1.5 Counting-Based Search

Most of the search heuristics for CSPs depend on the information gathered from variables during the search, like the domain size of the variable and the impact of variables. Sometimes it is hard to decide which would be the first variable to branch on if we use the individual variable information to guide the search. For example, if we search on the size of variables, then it may happen at the initialization that every single variable has the same domain size. Considering the structure of CSPs, some researchers proposed search heuristics using more global information. Pesant et al. [3] detail CBS for CSPs, using the information from constraints. Here it is necessary to introduce *solution count* and *solution density*, because they are the basic concepts for CBS. CBS resembles more a succeed-first principle, the variable that has the maximum solution density will be selected first and the value achieving that maximum solution density will be assigned to the variable.

Definition 2.1 (solution count) The number of solutions for a given constraint c taking into account the domain of the variables in its scope $(x_1, \ldots, x_n)$, can be represented as *solution count* $\#c(x_i)$.

Definition 2.2 (solution density) Given variables $(x_1, \ldots, x_n)$ in the scope of c with finite domain$(D_1, \ldots, D_n)$ and the value $v \in D_i$, we have the formula:

$$\sigma(x_i, v, c) = \frac{\#c(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_n)}{\#c(x_1, \ldots, x_n)}$$

representing the *solution density.* The solution density shows the probability of variable-value assignment $(x_i = v)$ being part of a solution to c.

CBS relies on the solution density to guide the search. Pesant et al. [3] searches the highest solution density for each variable and each value. Algorithm 1 is adapted from *maxSD* [3].

```
max = 0;
foreach constraint $c(x_1, \ldots, x_n)$ do
    foreach unbound variable $x_i \in \{x_1, \ldots, x_n\}$ do
        foreach value $v \in D_i$ do
            if $\sigma(x_i, v, c) > max$ then
                $(x^*, v^*) = (x_i, v)$;
                $max = \sigma(x_i, v, c)$;
            end
        end
    end
end
return branching decision "$x^* = v^*$";
```

Algorithm 1: The maximum solution density search [3]

In CBS, we choose the variable based on the *solution density* among all the variables and constraints and the value that has the maximum support count among all the values. The support count is collected from *propagate* (a method used to remove inconsistent values) of the compact table, that is why we said the data collection is on the constraint level. CBS selects the global data on the constraint level that can avoid locally optimal. CBS is applied to many constraints, like *element*, *Alldifferent* [4], *global cardinality constraints(gcc)* [3], *regular* [5] and *knapsack* [5]. For this thesis, CBS will be applied on the *compact table* constraint.

2.1.6 Value Heuristic

The variable ordering heuristic search chooses the next variable to branch on for CSPs, the value ordering heuristic is trying to find which value will be assigned to the variable. To find a solution for CSPs efficiently, there are a lot of heuristic searches proposed, such as WDEG, IBS and ABS. Most of the heuristic search techniques for CSPs are mainly focused on variable heuristic, although there are some approaches focused on value heuristic searches. For example, IBS achieved both variable and value heuristics. Some researchers explored the value-only heuristic and proved the performance of their methods is also efficient. Smith and Sturdy [13] defined a value heuristic search that works well on some constraints and consumes less time to find all the solutions for CSPs. A look-ahead value heuristic method [14] proposed has good performance on hard CSPs.

2.2 Table Constraints

A table constraint C_{CT} is a set of tuples of arity r, and the set of tuples is called a *table*. We used $rel(C_{CT})$ to denote a table. The scope $scp(C_{CT})$ of the table constraint is an ordered list of variables $(x_1, \ldots, x_r)$. The arity r of the table constraint is the size of the scope. The table constraint C_{CT} is satisfied if and only if the value taken by each variable in its scope corresponds to a tuple in the relation $rel(C_{CT})$. The table constraint supports a value $a \in D(x_i)$ if and only if we can find a valid tuple T in the relation $rel(C_{CT})$ such that $T(x_i) = a$.

Compact Table (CT) is an efficient implementation of the table constraint which is combining the bit operations with enforced GAC on the table constraint. CT uses bit sets to maintain the supports for each variable-value pair. The calculation between the valid tuples and the tuples' masks use bit-wise operations. The inconsistent values are removed from the domain during the propagation.

2.2.1 Reversible Sparse Bit Set

Compact table [2] combines an efficient general arc consistency (GAC) method with table constraint that uses a special data structure named *reversible sparse bit set*. The reversible sparse bit set uses some techniques to accelerate the processing of data. The compact table implementation is the default table constraint in the Oscar library. Lecoutre and Vion [15] introduced the bit operations to speed up the algorithm looking for support faster. Lecoutre et al. [16] presented an efficient Generalized Arc Consistency (GAC) algorithm to reach valid tuples to find out supports. The compact table uses GAC to filter invalid tuples.

A *bit set* (also called bit vector, bit array, or bit map) is a data structure that maps an entire domain in an array of binary values $\{0, 1\}$. If data appears (1) in the *ith* bit of the n bits bit set, then the value i is part of the domain. Else, if the *ith* bit is set to 0 then the value i is not part of the domain. For example, if we want to represent the set of odd numbers smaller than 10 ($\{1, 3, 5, 7, 9\}$) a bit set of 10 bits could represent it and the result would be 0101010101.

Sparse set is a data structure studied by Briggs and Torczon in [17]. The sparse set contains the size $size_D$ of the domain D and two arrays $dense_D$ and $sparse_D$. The $dense_D$ maps into the $sparse_D$ by the values' indices. For example, for the set $\{5, 7, 1\}$, the value 5 of $dense_D$ will be at the $5th$ index of the $sparse_D$. In Figure 2.1, $dense_D$ is split into two parts by $size_D$; the first part of $size_D$ is considered to be part of D; the second part (gray squares) is considered to be deleted. The sparse set is an efficient data structure for the usual operations; it costs constant time by moving $size_D$ and change the maps between $dense_D$ and $sparse_D$ to add and delete members, and to test membership.

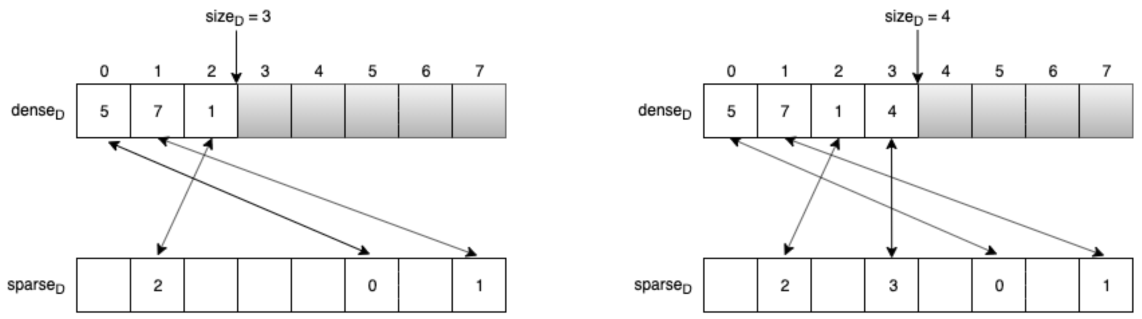

Figure 2.1 Representation of $spase_D$ and $dense_D$ before and after inserting value 4

The compact table maintains the supports for each variable-value pairs. As Briggs and Torczon in [17] presented, the *Reversible Sparse Bitset* is composed of several elements in the set that are reversible, the sparse set and the dense set. A sparse array stores the position of each element of the set in a dense array. Size indicates the number of elements in the current set. Schaus et al. [18] introduced the instrumented sparse bit set to represent the domain of variables which allows accessing the delta changes without cost. The compact table uses reversible sparse set for the $currTable$ which allows the solver to go back to the previous state when the solver backtracks. The reversible sparse bit set data structure achieves the restoration of the values from dense data to sparse data in constant time upon backtracking.

Figure 2.2 shows how to remove value 7 and when we want to restore the domain. First, we need to identify the position of 7 in the domain of the sparse set and dense set. Second, we need to identify the value in the last position of the dense set. Third, we need to exchange the position of these two values and update the map. Lastly, we decrement the size from 3 to 2. The process of restoring the values costs constant time in Oscar. When a failure occurs and the algorithm backtracks, we want to go back to the previous state and the deleted value comes back to the variable's domain. How to restore deleted values when backtracking occurs? We only need to restore the size to the previous size, so the deleted value will be returned to the domain.

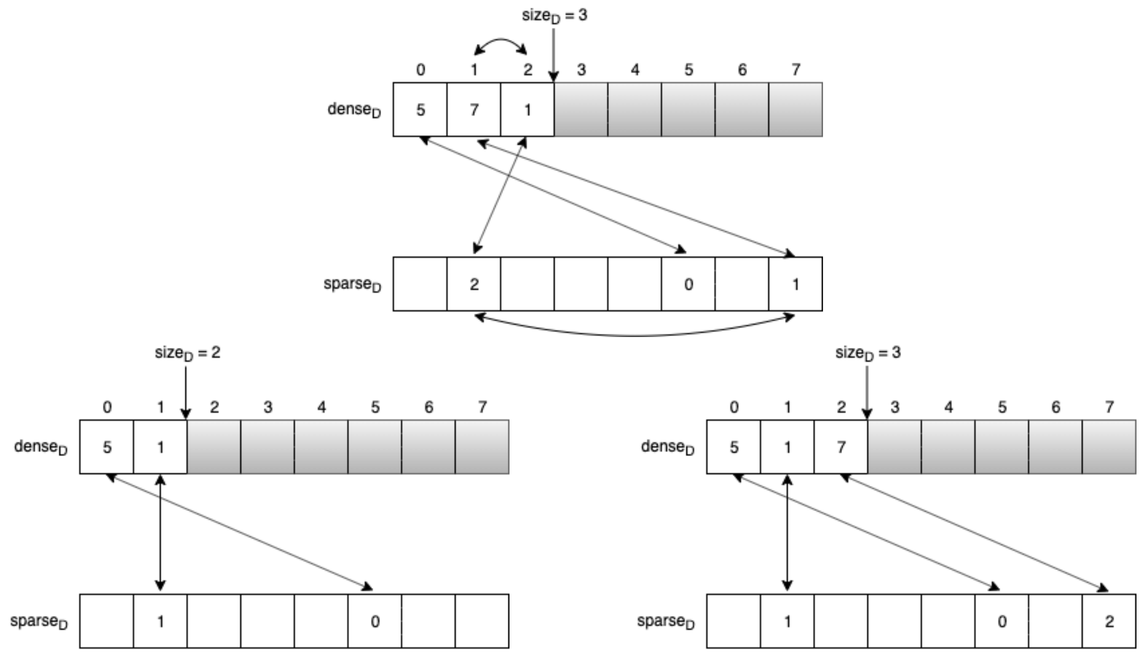

Figure 2.2 Operation performed to remove 7 from set and set $\{5, 7, 1\}$ after the restoration of $size_D$ (from top to left to right)

2.2.2 Representation of Reversible Sparse Bit Set

Table 2.1 shows an example of the indexed tuples of the table. Table 2.2 shows the corresponding bit sets after the initialization of the table constraint C_{CT} given in Figure 2.3.

$$
\begin{gathered}
scp(C_{CT}) = \{X(1), X(2), X(3)\} \\
D(X(1)) = \{2, 4\} \\
D(X(2))) = \{1, 5, 6\} \\
D(X(3)) = \{3, 6, 7\} \\
rel(C_{CT}) = \{(2, 1, 6), (2, 1, 7), (2, 5, 6), (2, 5, 7), \\
(4, 1, 6), (4, 1, 7), (4, 3, 7), (4, 5, 6), (4, 6, 7)\}
\end{gathered}
$$

Figure 2.3 Initialization of ExampleTable

The *currTable* is a reversible sparse bitset data structure (it is either 1 or 0) and some bits change from 1 to 0 if the assignment is invalid or backtrack if the bit set changed from 0 to 1. If the *AND* calculation result between *currTable* and supports for a variable-value pair is all 0, the value should be removed because it does not have *support*, and a backtrack will occur.

In the initialization of the compact table C_{CT}, a static bit-set $supports[X, a]$ $(X \in scp(C_{CT}) \cap a \in dom(X))$ for each variable-value pair (X, a) was computed.

Table 2.1 The indexed tuples

T	X(1)	X(2)	X(3)
1	2	1	6
2	2	1	7
3	2	5	6
4	2	5	7
5	4	1	6
6	4	1	7
	4	3	7
7	4	5	6
8	4	6	7

Table 2.2 The corresponding bit-set

currTable	1	1	1	1	1	1	1	1
supports [X(1),2]	1	1	1	1	0	0	0	0
supports [X(1),4]	0	0	0	0	1	1	1	1
supports [X(2),1]	1	1	0	0	1	1	0	0
supports [X(2),3]	0	0	0	0	0	0	0	0
supports [X(2),5]	0	0	1	1	0	0	1	0
supports [X(2),6]	0	0	0	0	0	0	0	1
supports [X(3),6]	1	0	1	0	1	0	1	0
supports [X(3),7]	0	1	0	1	0	1	0	1

Tuple $\{4, 3, 7\}$ is invalid since initialization because $3 \notin D(X(2))$. Value 3 will be removed from $D(X(3))$, because it is not in any relation $rel(C_{CT})$ of CT. The $currTable$ is a reversible sparse bit set data structure, and it is updated during the propagation. When the tuple is invalid, the $currTable$ bit will change from 1 to 0. When the search backtracks, the $currTable$ bit will change from 0 to 1. We need to mention that the supports will not change during the search.

2.2.3 Variants of Table Constraints

Some efficient implementations of table constraints for different situations were presented recently, such as *compact table*, *short table*, and *smart table*. *Compact table* was introduced in the beginning of Section 2.2.

- *Short table* [19] presented by Verhaeghe et al. extends the compact table to the negative table that means table contains conflicts and short table which contains symbol $*$. For example, $\{(c, *, a), (a, b, c), (b, c, b)\}$ is a negative short table for three variables.
- *Smart table* [20] is the table constraint that contains simple arithmetic constraints. Verhaeghe et al. [21] also extend the compact table to basic smart table, the smart

table contains some arithmetic symbols like $\leq$ and $\geq$, these symbols can make the table constraint adapt many CSPs. For example, $\{(\neq a, *, c), (c, \leq b, \neq a), (< c, b, \neq b)\}$ is a basic smart table for three variables.

- Ingmar et al. [22] presented dynamically compact bit sets and shared tables to reduce the runtime and memory usage of the compact table.

CHAPTER 3 COUNTING-BASED SEARCH IN OSCAR

In this chapter, there are three main sections. In Section 3.1, we introduce the big picture of Oscar, such as the variables, methods in the compact table and branching methods. In Section 3.2, we describe the changes we made in Oscar to adapt counting-based search. In Section 3.3, we introduce the counting-based search we created in Oscar based on the changes in Section 3.2, and we use a UML diagram to show the connections after changes.

3.1 Basic Concepts in Oscar

Oscar [1] is a dense and complicated library implementing a constraint programming solver. It is an open-source library written in Scala. We cannot explain thoroughly all the classes and packages of Oscar. In this section, we will only introduce the classes and methods that we will use in Section 3.2. The UML graph will show the basic structure of Oscar and the relationships between all the classes and packages in Oscar.

3.1.1 The Bigger Picture

It is important to understand each small method of the important classes of a project but it is also crucial to understand how the project works as a whole. To explain this, we decided to create a UML graph(see Fig. 3.1) of the important parts. Some keywords in UML are presented as the following sub-points.

In the *Oscar-cp-examples*, there is some information we want to introduce, such as the interface with search and constraint.

- *binaryIdx* is a function in the trait *Branchings* that extends the trait *BranchingUtils*. Binary search can be defined with the custom variable-value heuristic because the function *binaryIdx* has an instantiation class *BinaryBranching*. We should use this class and create our heuristic search specified with the parameters.
- *table* is a keyword to access the trait *Constraints*. In this trait, there is a function table that extends the compact table.

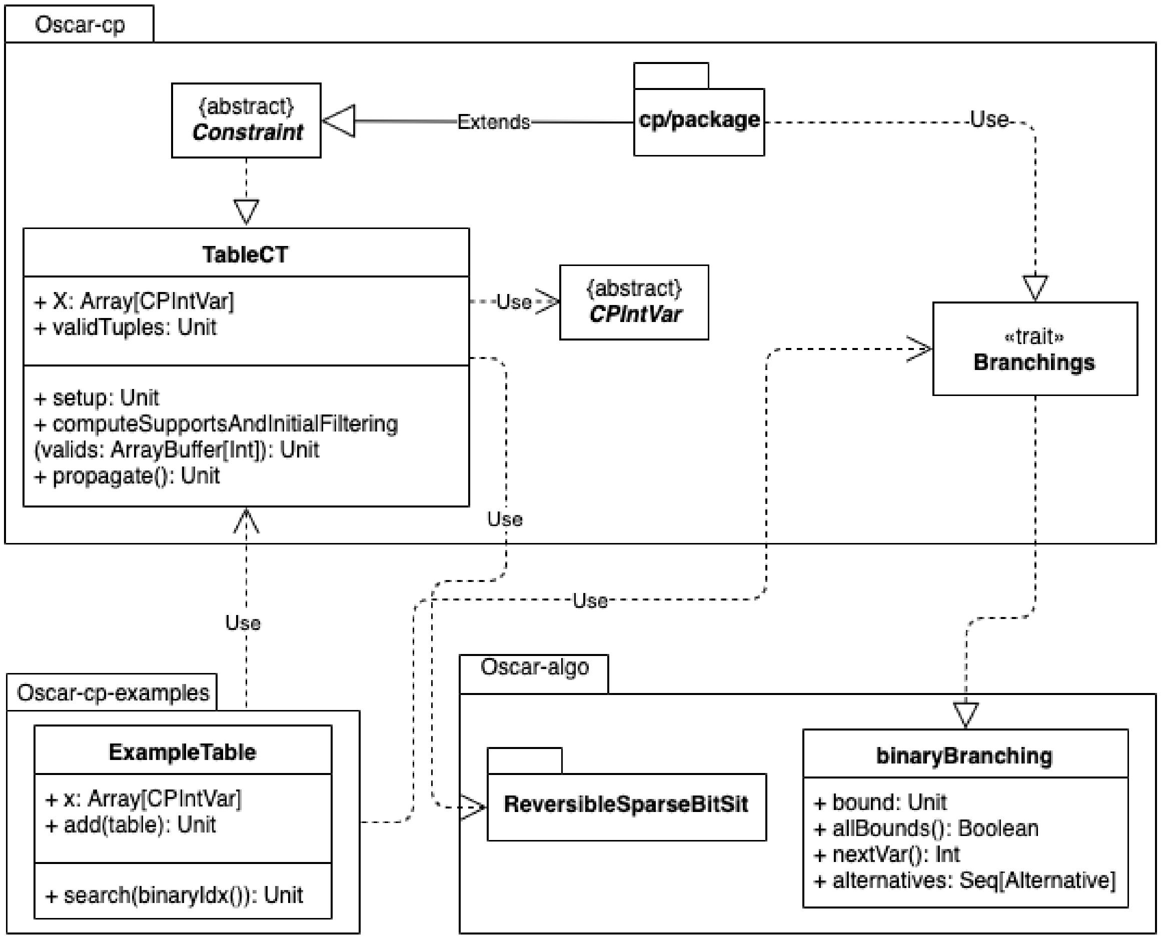

Figure 3.1 UML of part of Oscar

3.1.2 The *CP* package

In Oscar, the *cp/package* gives some useful features for modeling. There are some common types such as *CPSolver*, *CPIntVar* or *CPIntervalVar*. In the package, Oscar also defines several commonly-used implicit conversions and provides other functions such as *regret method* (returns difference in terms of cost between the second smallest and the smallest value in the domain). This package also defines the *CPModel* trait, which provides the user with an implicit solver called *CPSolver*.

3.1.3 Variables in Oscar

CPIntVar is an abstract class integrating some basic methods, like the domain size and the minimum value of the variable (see Fig. 3.2). Variables in the compact table are an array of *CPIntVar* hence variables can use the methods in the *CPIntVar* class. *CPIntVar* extends

CPVar. CPVar is an abstract class and it contains definitions of *store*, *isBound*, and *name*.

```
abstract class CPIntVar extends CPVar with IntVarLike {
   \\return true if the variable is bound to value v, false if variable
   \\is not bound or bound to another value than v

   def isBoundTo(v: Int): Boolean

   \\return  the size of the domain
   def size: Int

   \\return  the minimum value in the domain
   def min: Int
   ......
}
```

Figure 3.2 CPIntVar class

3.1.4 Methods in Compact Table

In the CT, there are several methods that we will use for CBS. The *setup* shows the basic organization and processing steps of the CT. The setup calls the *ComputeSupportsAndInitialFiltering* method and the *propagate* method. Some of the methods are used during the setup of the constraints, like the *collectValidTuples* and *isTupleValid* method.

- *ComputeSupportsAndInitialFiltering* retrieves the current valid tuples and collects the supports from the tuples. In this method, the final support *bit Sets* are created and any value that is not supported is removed.
- *Propagate* is the core method for all the constraints and it will be introduced later in Algorithm 4. When the constraint checks domain consistency, the value causing inconsistency will be removed. For each variable-value pair, propagate checks if there is at least one valid tuple remaining.
- *collectValidTuples* is a method called by *ComputeSupportsAndInitialFiltering.* This method retrieves the valid tuples from the table and stores their index in *validTuplesBuffer.*
- *isTupleValid* checks if a tuple is valid.

3.1.5 Branching Methods

Oscar offers users several state-of-the-art heuristic searches, like the Conflict-Ordering Search [23] for scheduling problems. We will use binary branching to specify our variable and value ordering heuristics. The variable ordering heuristic first selects the unbound variable, such as the one with the smallest domain. The value ordering heuristic determines the selection order of values, such as selecting the minimum value in the domain of the variable.

Binary branching is binary choices for variables; branching will occur when propagators filter values and the constraint does not allow you to filter anything anymore. Once the propagator in the compact table constraint removed the values that caused inconsistency, then the branching method will give a decision as defined. In *cp/packages*, we will not only compute and select the information from the compact table but also use the stored data from the compact table to guide the binary branching.

The binary branching class has four methods: *bound*, *allBounds*, *nextVar* and *alternatives*.

- *bound and allBounds* will check the availability of variables and values for branching.
- *nextVar* is used to define what is the next variable we should search on. To do so, *nextVar* iterates on every variable available. A variable is available if it has not been bound before. Then it will use a feature, for example, the domain size, to order every variable. Finally, we decide to search for the best variable from the ordered sequence.
- *alternative* checks if a solution is still possible for a problem. Then it searches the next variable and value to be assigned. Finally, it declares how to assign the variables in the child nodes.

In Figure 3.3, x are the decision variables to branch on; *i => x(i).size* is the variable heuristic, it returns an ordered value such that the solver will choose the variable that has the smallest domain for an index i in variables; *i => x(i).randomValue* is the value heuristic, it returns the random value in the domain of $x(i)$ assigned on the left branch and removed on the right branch.

```
search(binaryIdx(x,i => x(i).size,i => x(i).randomValue))
```

Figure 3.3 CPIntVarOps class

Binary branching is the method we will use to implement our counting-based search. In binary branching class, we return a list containing the left and right branches. The left branch assigns to a variable a value in its domain, whereas the right branch removes a value of the domain we searched on (which is the value taken by the left branch). Then, when we see that the solution on the left branch is not possible, we backtrack and go directly to the right branch. Figure 3.4 describes the backtrack scheme in Oscar. Each number represents a step. The loops represent propagation (steps 2 and 6).

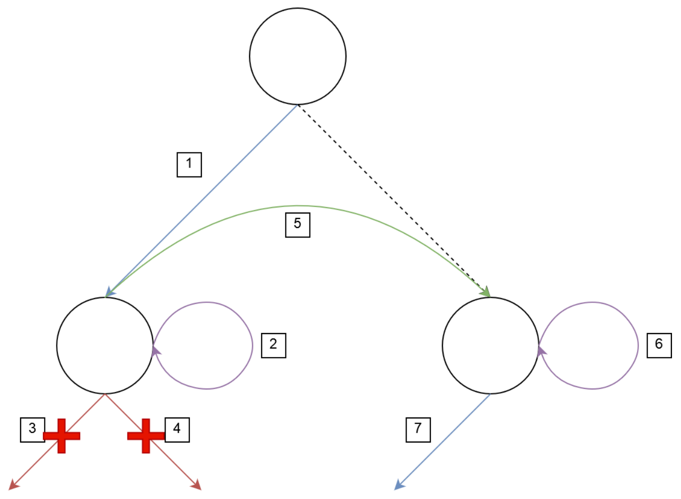

Figure 3.4 Backtrack in Oscar

Once we backtrack, some values are brought back to a previous state that was registered in the memory. This allows us to be more efficient. When the solver backtracks, we restore the size of the domain to its previous value and the values will be back in the domain. The left branch pushes the current state and the right branch pops the previous state when the solver backtracks and the domain will be restored.

3.2 Changes in Oscar

We introduce *supportCount* in Oscar because we need to use *supportCount* to calculate solution density. Second, we propose the modifications and algorithms to update the *supportCount*.

3.2.1 The *supportCount* in Oscar

supportCount is the number of ones in the bit set tuple for each variable-value pair. The first thing we need to consider is how to collect *supportCount* for CBS. Using bit-wise AND operation between *currTable* and supports of the variable-value pairs, then the sum of bits in the result is *supportCount*. Definition 3.2.1 is the definition of the *supportCount*:

Definition 3.2.1(supportCount)

$$supportCount = \sum (currTable) \cap (supports[x, v])$$

When we calculate the *supportCount* for each value, we really need to consider the views. A view is a way to represent a variable. It is used in *TableCT* to save space, for example if the domain is $D_1 = \{1, 3, 4, 5\}$ and the view is just an offset (like in *TableCT*), we will have an offset of 1 and the domain will be represented in the class as : $D_1 = \{0, 2, 3, 4\}$ Hence, during the calculation of our *supportCount* we really need to consider the offset of our view to place the count at the right place and to later search on the right value using the *supportCount*.

Table 3.1 shows the *supportCount* we get for the ExampleTable(see Figure 2.3).

Table 3.1 The *supportCount* for each variable

	v = 1	v = 2	v = 3	v = 4	v = 5	v = 6	v = 7	v = 8
X(1)	0	4	0	4	0	0	0	0
X(2)	4	0	0	0	3	1	0	0
X(3)	0	0	0	0	0	4	4	0

3.2.2 Plan of Modification

Now that we saw the bigger picture, we can ask ourselves which modifications we want to bring. First of all, our objective is counting-based search using solution density. So we will create our search methods to have the first sketch of them to use later. Now, we need to calculate the solution density for each of our variables and transfer them to our search algorithm. Then, we need to create our search algorithms.

```
binaryIdx(X, i => - X(i).varCountSearch, i => X(i).valCountSearch)
```

Figure 3.5 Binary branching with our newly created methods

The first thing we need to consider is where we should bring our changes. We decided to use *cp/package* as the main file we will modify as it can be accessed both by the compact table and the search. Hence, we will use this file as a bridge between the calculation of the *supportCount*, and the decision on a new variable-value pair to search on. Also, Oscar is complex, and limiting most of our modifications to only one file can bring the complexity of the changes to a minimum for someone that would like to use or continue our work. It is also important to notice that we could also have used the $CPIntVar$ class but we chose to use *cp/package* as $CPIntVar$ is an abstract class that is inherited by a lot of other classes. Modifying it would possibly bring a lot of problems on other projects of other users of Oscar but also on other subclasses of our project.

3.2.3 Type of Storage

In this part, we want to take a look at which kind of storage we want to use to store the *supportCount* of our variables. First of all, we decided that it should be storage easy to access and really efficient as it will be used multiple times in multiple methods. Hence, we decided to use a map as the storage because values (*supportCount*) would be accessible in a $O(1)$ complexity. Then we need to decide which information we need to store: of course, we need to store the *supportCount*, but we need to store it for each of our variables and also our constraints. To conclude, we decided to use a Map of Map (*mapcount*) using variables and constraints as indices for each map respectively.

3.2.4 Unique ID for Constraints and Variables

To fill a map, we need an argument that can be hashed as a key and that will represent a single member. But, because of views, as explained before, in each class the way to represent variables and constraints is not the same, then we cannot use the normal representation to access the *supportCount* and we need to find a unique ID to represent both our variables and our constraints.

Hash code is an excellent way to keep the consistency of variable storage and usage. Hash code comes from a Java class in Oscar, and the hash code will return the same integer regardless of which class a given variable is involved in during the execution.

3.2.5 Changes in Compact Table

The map including *supportCount*s and the unique ID for constraints and variables, we will call it *mapcount* from now on, needs to be maintained in a static object. When we search, we need the *mapcount* to be stable but we also need to update the *mapcount* when propagation occurs. If *mapcount* is not a static object, the content inside will be erased. In Scala, the singleton object is used as the static object. Hence, we need to store the *mapcount* as a singleton object. Table 3.2 shows the data structure of *mapcount*:

Table 3.2 The structure of mapcount

variable 1	constraint 1	supportCount
	constraint 2	supportCount
	......	......
	constraint m_1	supportCount
....	......	......
variable n	constraint 1	supportCount
	constraint 2	supportCount
	......	......
	constraint m_n	supportCount

The information we collect during the processing of code will not be available after the end of the process. To deal with this problem, we found the singleton variables in Scala. In Scala, an object is a class that can define static arrays, and then the information can be saved. This is one of the reasons why we do not process all our data directly inside *TableCT* (see Figure 3.1), as it will not be saved. If we use the same name for object and class in the same source file, the object and class are companions. We created a 2-dimensional hash table in the object class to store the information from *TableCT* and instances. In this section, we will introduce the different variables, arrays, and hash table, then we will introduce the whole design of the algorithm.

- *idvar* is the hash code of variables; it is one of the information we need to get from the compact table.

- *idconst* is from the instances, like *ExampleTable*. We get the constraint ID from the class *Constraint* and transfer the constraint ID to the compact table. *Idconst* is one of the parameters in the method we created.

- *supportCount* is a one-dimensional array that is defined in the compact table. Here, we used a two-dimensional hash table to store the *supportCount* for a given variable and constraint ID. The *supportCount* in the compact table will be updated during execution, and the *supportCount* will be stored in the hash table when the *supportCount* is updated.

- *mapcount* is defined in the Object class $CPIntVarOps$ it is a two-dimensional map. The hash table *mapcount* is a $HashTable[Int, [Int, Array[Int]]$ data structure. It contains every information needed to do the branching.
- *mapVarVal* is a two-dimensional hash table, the first field is the *idvar* and the second field is the index of the value in *supportCount* which has the maximum *supportCount*.
- *mapCons* is a $HashTable[Int, Array[Int]]$ data structure, it contains constraint ID and *supportCount*.
- *mapVarSD* is a two-dimensional hash table, the first field is the *idvar* and the second field is the solution density of each variable. The hash table $mapVarSD$ is a $HashTable[Int, Float]$ data structure.

We also decided to change our storage following some results. In fact, in the beginning, our map was indexed first by constraint and second by variable. This meant that for one variable we had to loop on all the constraints to determine the solution density. In the end, if we have 100 constraints and our variable was involved in only 10 of them we would loop 100 times anyway. We changed it considering that we would loop more efficiently for each variable (10 times in our example).

We use a small example composed from a limited number of variables and constraints to show how our structures are used in the runtime of our algorithm. To simplify things, the *hashCode()* we obtain from the variables and constraints are represented respectively by *Vi* and *Ci*. In Figure 3.6, you can find the example.

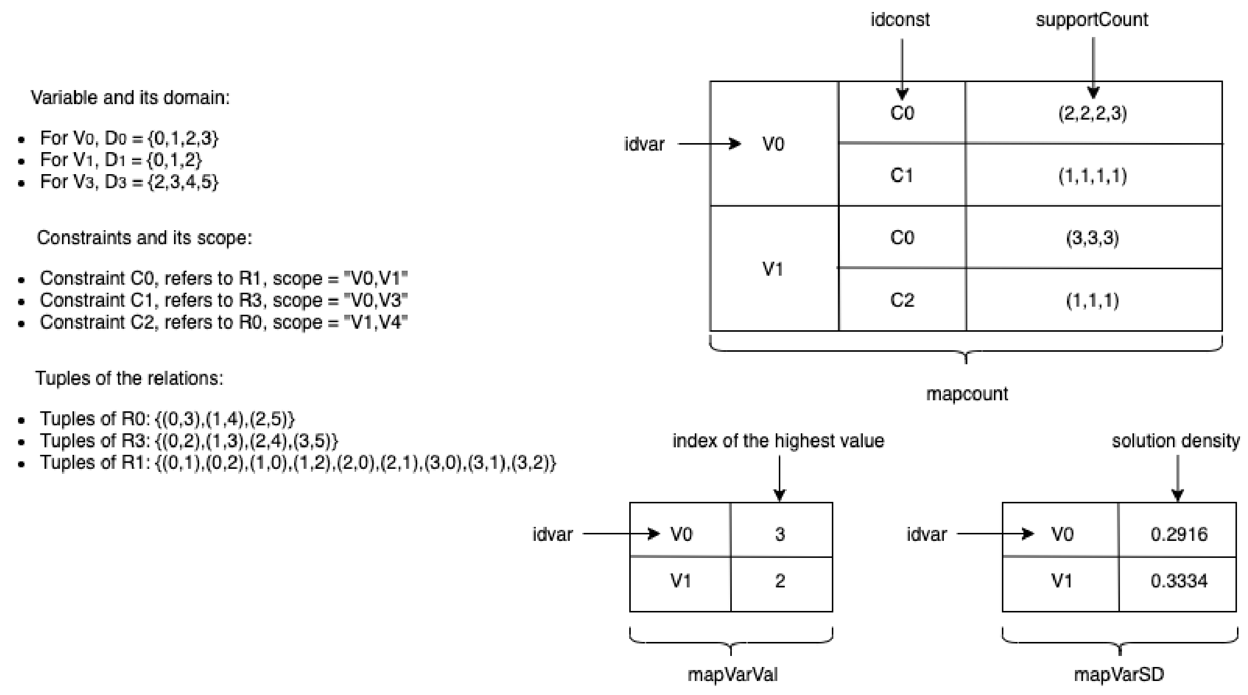
Variable and its domain:
For V0, D0 = {0,1,2,3}
For V1, D1 = {0,1,2}
For V3, D3 = {2,3,4,5}
Constraints and its scope:
Constraint C0, refers to R1, scope = "V0,V1"
Constraint C1, refers to R3, scope = "V0,V3"
Constraint C2, refers to R0, scope = "V1,V4"
Tuples of the relations:
Tuples of R0: {(0,3),(1,4),(2,5)}
Tuples of R3: {(0,2),(1,3),(2,4),(3,5)}
Tuples of R1: {(0,1),(0,2),(1,0),(1,2),(2,0),(2,1),(3,0),(3,1),(3,2)}
idconst
supportCount
idvar
V0
C0
(2,2,2,3)
C1
(1,1,1,1)
V1
C0
(3,3,3)
C2
(1,1,1)
mapcount
index of the highest value
idvar
V0
3
V1
2
mapVarVal
solution density
idvar
V0
0.2916
V1
0.3334
mapVarSD

Figure 3.6 Example of structures' usage

In *cp/package*, we created an object named *CPIntVarOps*, it has the same name as the class in which we will create our method to update the *supportCount*. In object *CPIntVarOps*, we defined a map named *mapcount* to store the variable ID, constraint ID and *supportCount*, while we will define *changeCount* method to update them in the class *CPIntVarOps*. The *TableCT(compact table)* will call the method *changeCount* to fill the *mapcount* to update the *supportCount*. In the algorithm below, the *mapCons* is used to store the mapped *idcons* to *supportCount* and the *CPIntVarOps.mapcount* will keep the information we selected. We removed the *SD(solution density)* and value out of *mapVarSD* and *mapVarVal* respectively because once we propagate on value, we know that our best value saved is no longer valid. Also, it allows us to limit the space taken in memory because a value that will become bound through propagation will update its count and will not be kept in the memory anymore.

```
  input: idvar, it is the variable ID for hash table index
         idconst, it is the constraint ID for hash table index
         supportCount, it is an array for supportCount
1 CPIntVarOps.mapVarSD.remove(idvar) ;             /* remove the variable */
2 CPIntVarOps.mapVarVal.remove(idvar) ;               /* remove the value */
  /* retrieve the mapCons contained in this idvar if it exists, not to
     overwrite it                                                      */
3 if CPIntVarOps.mapcount.get(idvar) != None then
4     mapCons = CPIntVarOps.mapcount(idvar)
5 end
  /* adding new element in mapCons                                     */
6 mapCons = mapCons + (idconst -> supportCount);
7 CPIntVarOps.mapcount = CPIntVarOps.mapcount + (idvar -> mapCons);
```

Algorithm 2: Update of *supportCount*(changeCount)

In Algorithm 2, we can see the time complexity is $O(1)$. Once we defined the algorithm *changeCount*, we need to get the information from *TableCT* and Constraint. In the *initial-Filtering* and the *propagate*, we need to call the method *changeCount*.

```
input: initCount, the count in the initialization
       maxValue, declaration of the maximum value in the domain of a variable
       supCount, the array of supportCount
foreach varIndex <- variableValueSupports.indices do
    supCountSize = maxValue + 1 ;                /* the size of supportCount */
    foreach valueIndex <- variableValueSupports(varIndex).indices do
        if varValueSupports(varIndex)(valueIndex).nonEmpty then
            variableValueSupports(varIndex)(valueIndex) = new
             validTuples.BitSet(varValueSupports(varIndex)(valueIndex));
            initCount =
             validTuples.intersectionCount(varValueSupports(varIndex)(valueIndex));
            supCount(valueIndex + offsets(varIndex)) = initCount ;        /* add
             initCount in the supCount array, offsets represents the
             offset of the view */
        end
        else
            x(varIndex).removeValue(valueIndex)
        end
    end
    if supCount.max > 0 then
        x(varIndex).changeCount(supCount, X(varIndex).hashCode(), getConsID)
    end
end
```

Algorithm 3: Call the changeCount in ComputeSupportsAndInitialFiltering

In Algorithm3, if we call *changeCount* before we removed values, the *supportCount* will have the remaining values which were deleted during the propagation. Some variables do not have any valid supports, so we add the condition to only update the variables which have supports. We need to mention that the blue part is the changes we made and it is the same for Algorithm 4.

```
input:  unBoundVars, the unbound variables
        unBoundVarsSize, the number of unbound variables (ReversibleInt type)
        unBoundVarsSize_, the value of unBoundVarsSize
        Index, the value of unBoundVarsSize
        domainArray, the domain of variable
        domainArraySize, the domain size of variable
        propCount, the count in the propagate
        maxValue, the maximum value in the domain of variable
        supCount, the array of supportCount
while $Index > 0$ do
    Index -= 1;
    varIndex = unBoundVars(Index);
    if x(varIndex).isNotBound then
        supCountSize = maxValue + 1 ;          /* the size of supportCount */
        while $domainArraySize > 0$ do
            domainArraySize -= 1;
            value = domainArray(domainArraySize);
            if !validTuples.intersect(varValueSupports(varIndex)(value)) then
                x(varIndex).removeValue(valueIndex);
            end
            propCount =
              validTuples.intersectionCount(varValueSupports(varIndex)(value));
            supCount(valueIndex + offsets(varIndex)) = propCount ;      /* add
              propCount in the supCount array */
        end
        if $supCount.max > 0$ then
            x(varIndex).changeCount(supCount, X(varIndex).hashCode(), getConsID)
        end
    end
    if x(varIndex).isBound then
        unBoundVarsSize_-= 1;
        unBoundVars (Index) = unBoundVars (unBoundVarsSize_);
        unBoundVars (unBoundVarsSize_) = varIndex;
    end
end
```

Algorithm 4: Call the changeCount in propagate

In Algorithm 4, we update the SupportCount during the propagation, basically we loop on each constraint not assigned yet and we calculate it is SupportCount. We use the following notations for the rest of the chapter : n is the number of variables, m is the number of constraints, w is the number of values in the domain of each variable and t is the number of tuples. The time complexity of Algorithm 3 and Algorithm 4 is $O(t \times n)$ because this is the number of times needed to call Algorithm 2 to fill the *mapcount*, and it will be called for each constraint m.

3.3 Search in Oscar

CBS uses the information of the *mapcount* to guide the search. We will introduce the principles of CBS and design the algorithms of CBS. A UML figure will show the organization of Oscar after changes.

3.3.1 Principle of CBS

Support is essential information in CSP; the *supportCount* for variable and value pairs can guide the search. There is a different combination of a selection of variable-value pairs. The counting-based search uses the information at the constraint level and we collect the *supportCount* from the compact table. The variable with the maximum solution density will be chosen first, and the value with the maximum *supportCount* will be selected.

3.3.2 Variable Selection

The variable search uses the maximum solution density to guide the search. Why it chooses the maximum solution density, not the maximum *supportCount* to conduct the variable search? There are two different reasons for that, first of all, most of the time the maximum *supportCount* would be the same for a lot of variables and will not be discriminative enough. Second of all, we want in the end to narrow down our search tree as fast as possible to make the search finish faster. Let's take an example in which we have two variables and one constraint for a problem. The first variable has 10 domain values with an average *supportCount* of 1.5 and a maximum *supportCount* of 3 whereas the second variable has 5 elements with an average *supportCount* of 1.5 too and a maximum *supportCount* of 2. In this situation, it is better to search on the second variable first even if the maximum *supportCount* is lower for the simple reason that we will go in the depth of the tree faster and so find faster if the problem has a solution or not. Hence, the probability to find a faster solution should consider how many elements each variable has (3/(1.5*10) compared to 2/(1.5*5)).

```
1  SD = 0f ;                        /* the solution density in float type */
2  SDTest = 0f;
3  mapCons = CPIntVarOps.mapcount.getOrElse(x.hashCode(), Map[Int,Array[Int]]) ;
     /* map of variable to (map of constraint to supportCount) */
4  foreach variable <- mapCons do
5      supportCount = mapCons._2 ;               /* get the correct supcount */
6      SDTest = supportCount.max.toFloat / supportCount.sum.toFloat;
7      if SDTest > SD then
8          SD = SDTest ;
9          CPIntVarOps.mapVarVal = CPIntVarOps.mapVarVal + (x.hashCode () ->
             supportCount.indexOf(supportCount.max)) ;   /* save the value which
             has the maximum supportCount */
10         CPIntVarOps.maxVarSD = CPIntVarOps.mapVarSD + (x.hashcode() ->
             SD);
11     end
12 end
13 if SD == 0f then
14     SD = CPIntVarOps.mapVarSD.getOrElse(x.hashcode(),0f)
15 end
16 return SD;
```

Algorithm 5: Variable search based on solution density

We calculated the solution density using the $supportCount.max.toFloat/supportCount.sum.toFloat$. The solution density is corrected only if there are no remaining values in the *supportCount* array. For example, if the maximum *supportCount* is removed during propagation, the *supportCount.max* will be the wrong value. We did a filter to test if we have the remaining values in the *supportCount*. $(x.toArraycollectsupCount).sum$ and $(x.toArraycollectsupCount).max$ are used to collect the *supportCount* for the values in the domain of variable. We proved that $(x.toArraycollectsupCount).sum$ is equal to $supportCount.sum$ and $(x.toArraycollectsupCount).max$ is equal to $supportCount.max$. In Algorithm 5, the time complexity is $O(m)$ but called $O(n)$ times in $binaryBranching$ class.

3.3.3 Value Search

We explained the method to search the variables, and then we need to introduce the approach to select the values for each variable. We used the *mapVarVal*(created in the variable selection) to store the value for which that maximum is achieved for each x, then *valCountSearch* (i.e., value selection) would return the stored value for x (the selected variable). We also took care of cases where all the variables with *supportCount*s have been associated, we then turn to a simple min or max algorithm for variables without any *supportCount*s.

```
1  mapCons = CPIntVarOps.mapcount ;            /* map of variable to (map of
     constraint to supportCount) */
2  valuereturn = CPIntVarOps.mapVarVal.getOrElse(x.hashCode(),-1) ;      /* value
     index to return at the end */
3  if valuereturn < 0 AND bound == "min" then
4      valuereturn = x.min ;      /* if the var was outside the TableCT and we
         decide to get the min */
5  end
6  else if valuereturn < 0 AND bound == "max" then
7      valuereturn = x.max ;      /* if the var was outside the TableCT and we
         decide to get the max */
8  end
9  CPIntVarOps.mapcount.clear();
10 CPIntVarOps.mapVarSD.remove(x.hashcode());
11 CPIntVarOps.mapVarVal.remove(x.hashcode());
12 return valuereturn ;
```

Algorithm 6: Value search

For Algorithm 6, the time complexity is $O(1)$ and it is called only once in *BinaryBranching*. We added some security in this algorithm in case no more variables had a supportCount (line 3 to 8). It is just used for variables that are added and not included in any constraint, we still need to choose the values for those variables to finish the algorithm. This happened only at the end of our tests instances, when we only add unbound variables left.

3.3.4 Improve the Efficiency of Variable Search

We clear the *mapcount* once a value is selected. We do it because we do not want to keep outdated information on the different variables and the *mapcount* will become smaller. Also, it prevents us from doing extra loops that are useless for the calculation. However, it happens that the *mapcount* becomes empty because we did not propagate on all the variables and the variables we did are now bounded. That is why we do not clear the *mapVarVal*, as it will contain the previous best values selected for each variable. We also added *mapVarSD* (see Figure 3.6) that contains the solution density for each variable in the previous search.

Hence, when a variable did not propagate we will keep those two values alive in the maps (not remove them) and will use them if we search on them. We added an "if" condition to trigger when the solution density is equal to zero in Algorithm 6, the solution density of each unbound variable but not propagated will be all considered and managed in this loop. The variable search will be improved because of the maintenance of *mapcount* and the computation of solution density will be improved.

3.3.5 A UML of Oscar After Changes

Figure 3.7 shows the whole structure of the design and implementation of this project. Compared with Figure 3.1, we can easily see that we added code in cp/package, including the object *mapcount* and *changeCount*, varSearch and valueSearch methods. From the relations among all the classes and packages in Figure 3.7, we can figure out that the *TableCT* and *binaryBranching* can call the methods in the *CPIntVar* and *cp/package*.

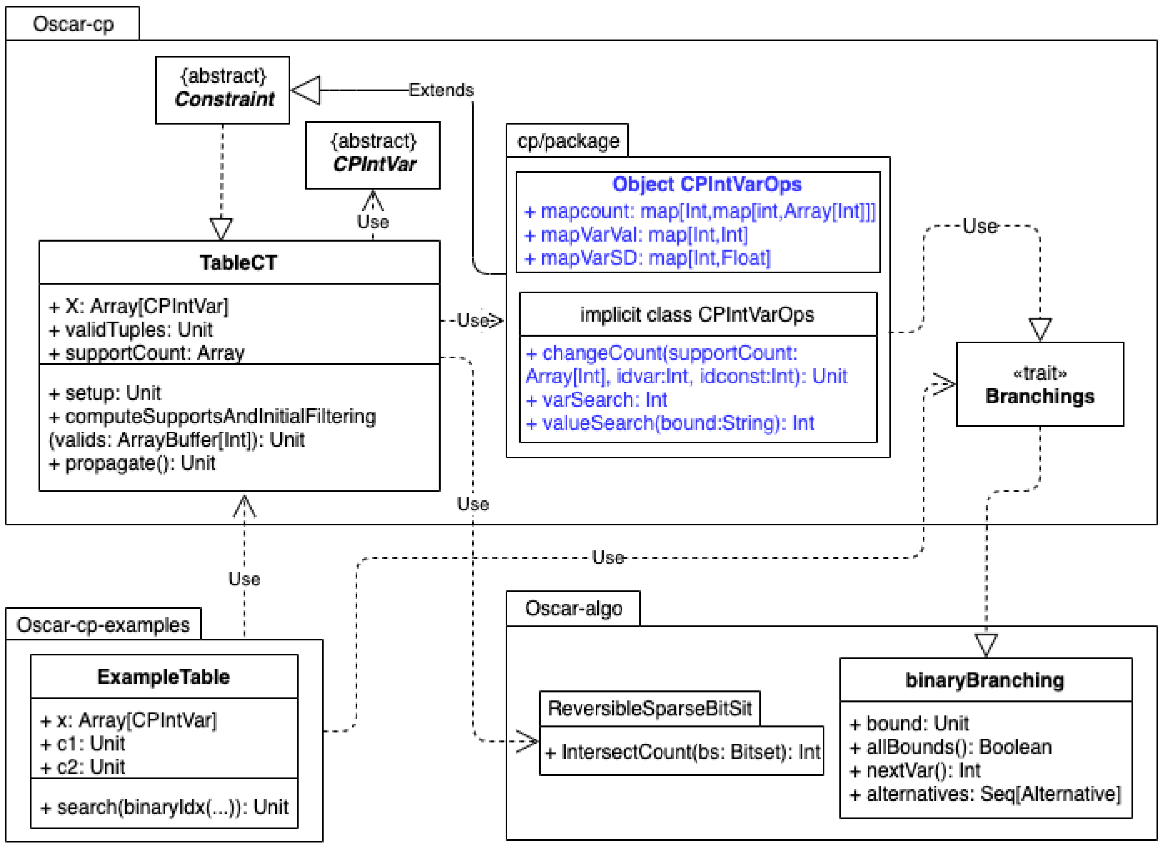

Figure 3.7 UML of CBS in the Oscar

3.3.6 Adding Arithmetic and Geometric Methods

The heuristic search *maxSD* [3] shows the counting-based search is better than other generic search methods in the experimental results. Sometimes, the *maxSD* will not make a good decision when there are many constraints associated with variables. To improve the performance of the algorithms, we need to try the arithmetic average and the geometric average of solution density that are the variants of *maxSD*. We decided to implement those two solutions to get an even more discriminative choice when it comes to which variable we should branch on.

```
sumSC = 0 ;                                          /* sum of supportCount */
SD = 0f ;                              /* the solution density in float type */
maxProb = 0f ;                                   /* the maximum probability */
mapCons = CPIntVarOps.mapcount.getOrElse(x.hashCode(), Map[Int,Array[Int]]) ;
foreach value <- variables do
    sumSD = 0f;
    numcons = 0;
    foreach constraint <- mapCons do
        if value > supportCount.size - 1 then
            SD = 0;
        end
        else
            SD = supportCount(value).toFloat / supportCount.sum.toFloat;
        end
        sumSD += SD;
        numcons += 1;
    end
    Probvi = sumSD / numcons.toFloat;
    if Probvi > maxProb then
        maxProb = Probvi;
        CPIntVarOps.mapVarVal = CPIntVarOps.mapVarVal + (x.hashCode() ->
          value);
        CPIntVarOps.maxVarSD = CPIntVarOps.mapVarSD + (x.hashcode() ->
          SD);
    end
end
if SD == 0f then
    SD = CPIntVarOps.mapVarSD.getOrElse(x.hashcode(),0f)
end
return maxProb;
```

Algorithm 7: Variable search based on the arithmetic average

```
sumSC = 0 ;                                          /* sum of supportCount */
SD = 0f ;                              /* the solution density in float type */
maxProb = 0f ;                                     /* the maximum probability */
mapCons = CPIntVarOps.mapcount.getOrElse(x.hashCode(), Map[Int,Array[Int]]) ;
foreach value <- variables do
    prodSD = 1f ;                         /* the product of solution density */
    numcons = 0;
    foreach constraint <- mapCons do
        if value > supportCount.size - 1 then
            SD = 0;
        end
        else
            SD = supportCount(value).toFloat / supportCount.sum.toFloat;
        end
        prodSD += (- log10(SD).toFloat) ;         /* the sum of the logs is the
          product of solution density */
        numcons += 1;
    end
    Probvi = prodSD / numcons.toFloat;
    if Probvi > maxProb then
        maxProb = Probvi;
        CPIntVarOps.mapVarVal = CPIntVarOps.mapVarVal + (x.hashCode() ->
          value) ;
        CPIntVarOps.maxVarSD = CPIntVarOps.mapVarSD + (x.hashcode() ->
          SD);
    end
end
if SD == 0f then
    SD = CPIntVarOps.mapVarSD.getOrElse(x.hashcode(),0f)
end
return maxProb;
```

Algorithm 8: Variable search based on the geometric average

Unfortunately for Algorithm 8, because we are multiplying multiple low float numbers, the memory cannot follow and we end up in a wrong calculation in the product of solution density that becomes equal to 0. To solve it, we could take the sum of the logs instead; this is a standard way to fix this. For Algorithm 7 and Algorithm 8, the time complexity is $O(m \times w)$ but this algorithm is called $O(n)$ times in *BinaryBranching*.

CHAPTER 4 EXPERIMENTAL RESULTS

In this thesis, CBS variants based on maximum solution density, the geometric average of solution density and the arithmetic average of solution density were introduced in Chapter 3. Our implementation is built using the Oscar [2] library, and the performance of the counting based search will be compared with other efficient search methods. Section 4.1 introduces the computer and benchmark we used to run our experiments. Section 4.2 describes the search algorithms we will run our algorithm against, the benchmark problems used and our experimental results. Section 4.3 shows which parameters affect our results.

4.1 Context

We tested 14 different problem classes and all the instances involved positive table constraints. All the instances that we selected to test our algorithms are available at `https://bitbucket.org/pschaus/xp-table/src/master/instances/` [2]. We used a MacBook Pro 3.1 with a 3.1 GHz Intel Core i5 processor and 8 GB 2133 MHz LPDDR3 memory; the operating system is macOS Mojave 10.14.5.

We set 1000 seconds as a time limit for all the instances. We report the time taken to find a solution or to prove that none exists, and the number of failed search tree nodes (i.e., backtracks).

4.2 Results

4.2.1 Tested algorithms

We used the following state-of-the-art heuristic branching methods:

- *maxSD*: it uses binary branching to select the variable with the largest solution density and the value for which that maximum is achieved for that variable;
- *maxAvgSD*: the variant of *maxSD* using the arithmetic average of solution density over constraints;
- *dom*: it uses binary branching to select the variable with the smallest domain as variable

ordering heuristic and chooses the random value in the domain of that variable as value ordering heuristic ;

- *dom/deg*: it uses binary branching to select the variable with the smallest domain over degree ratio as variable ordering heuristic and the random value in the domain as the value ordering heuristic.

4.2.2 The tested problems

We tested the following problems:

- *Quasigroup completion problem*: *aim* instances are created by Asahiro, Iwama, and Miyano [24]. It has *SAT* and *UNSAT* instances and *SAT* means the instance is satisfiable and *UNSAT* that it is not. They used the quasigroup completion problem to obtain their benchmark instances.
- *Renault Megane configuration problem*: the modified Renault problem comes from the real-word problem: Renault Megane configuration problem.
- *Kakuro puzzle problem* is a Japanese logic problem. You are given a grid which you need to fill in with numbers. You have clues on each line and column of the grid. The sum in the lines and columns should be equal to their corresponding numbers.
- *Traveling Salesman Problem* is a classic optimization problem. The principle is as follows: You are a salesman who needs to travel to different cities to sell your product, the optimization aims to find the shortest path in terms of cost-covering every city in the tour.
- *Dubois and A5*: We do not know what the *Dubois* and *A5* represented in terms of real-life problems. We will give the corresponding parameters for these two problems.
- *Nonogram* is a logic game to show hidden images. It uses the clues on the side of the grid to color cells or leaves them blank. For example, ”$4, 8, 3$” is a clue on the side of the grid, indicating that there are 4 filled squares, then 8 filled squares and finally three squares, each set of squares are separated by at least one space.
- *Langford number problem* is the 24th problem of CSPLib [25]. The aim of this problem is to sequence k copies of the integers 1 to n such that each occurrence of the number i is separated by i other numbers. For example, the sequence 41312432 would be a solution for k=2, n=4.

- *Crosswords problem* gives a grid to fill words from a dictionary with some clues.
- *Binary decision diagram* is a problem where we use a certain data structure to represent a boolean function. We decided to use the *bddsmall* instances because *bddlarge* is too big to be solved by our computer under our time-out threshold.
- *Easy generation of hard (satisfiable) instances* is random table constraints with a random scope. They are generated by Ke [26].

As we already described the time complexity for each algorithm in the Chapter3, here we want to conclude the calculation of the complexity of our whole algorithm when branching on a value for respectively $maxSD$ and $maxAvgSD$. We use the following notations for the rest of the chapter : n is the number of variables, m is the number of constraints, w is the number of values in the domain of each variable and t is the number of tuples. For $maxSD$, the time complexity is $O(n \times m \times t)$. The computational process of time complexity of $maxSD$ is as follows : $O(n \times m \times t)$ (from Algorithm 2, 3, 4) + $O(n \times m)$ (from Algorithm 5) + $O(1)$ (from Algorithm 6). Meanwhile, for $maxAvgSD$, the time complexity is $O(n \times m \times w \times t)$. The computational process of time complexity of $maxAvgSD$ is as follows: $O(n \times m \times t)$(from Algorithm 2, 3, 4) + $O(n \times m \times w)$ (from Algorithm 7) + $O(1)$ (from Algorithm 6).

Quasigroup Completion Problem

In *Aim50*, *Aim100*, and *Aim200*, the domain of each variable is binary, the number of table constraints is between 10 and 30 and in each table constraint the number of tuples is small and the arity is 3. *Aim50* has 50 variables and the number of constraints is small. *Aim100* has 100 variables and the number of constraints is medium and *Aim200* has 200 variables and the number of constraints is large.

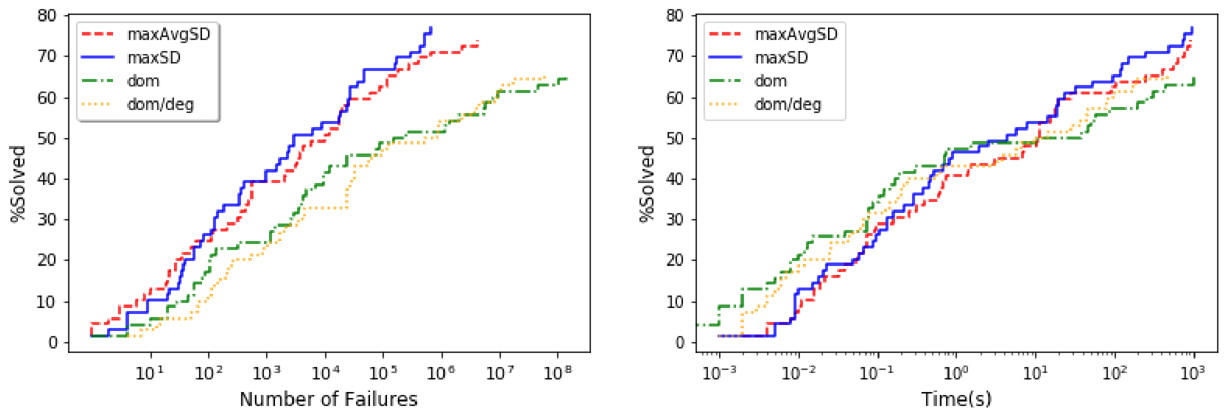

Figure 4.1 Percentage of Aim50 and Aim100 and Aim200 instances solved w.r.t. time and number of failures

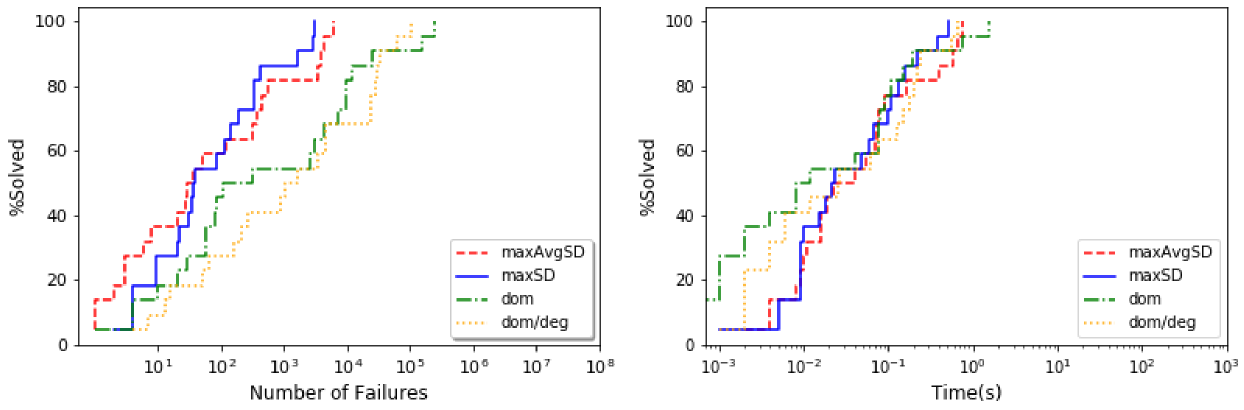

Figure 4.2 Percentage of Aim50 instances solved w.r.t. time and number of failures

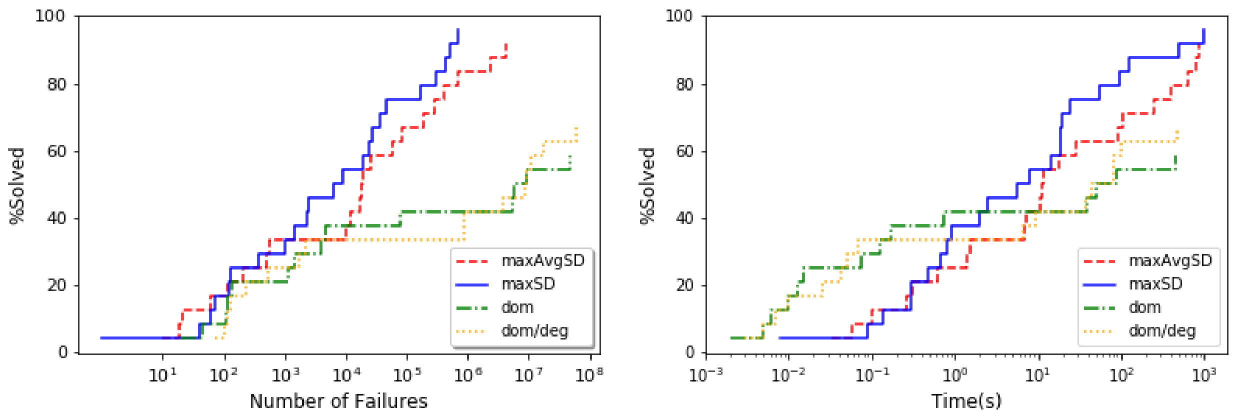

Figure 4.3 Percentage of Aim100 instances solved w.r.t. time and number of failures

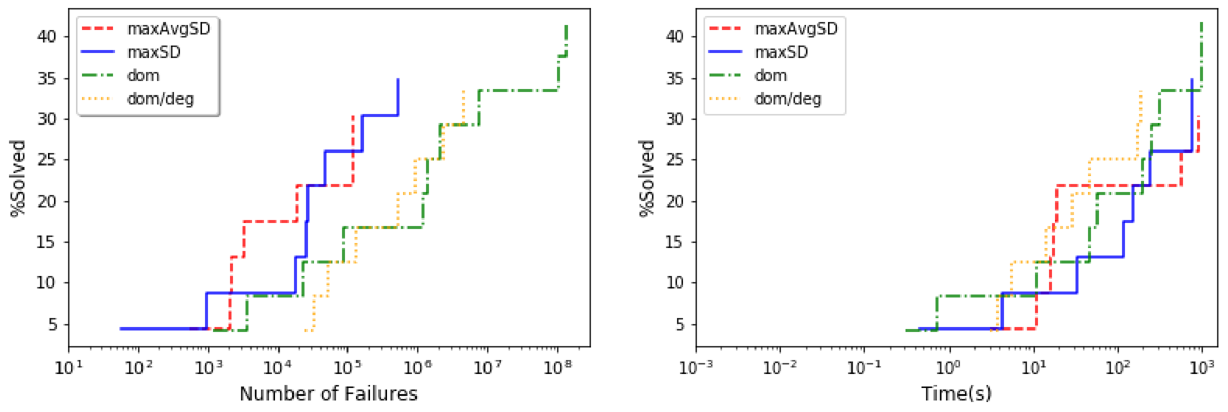

Figure 4.4 Percentage of Aim200 instances solved w.r.t. time and number of failures

In Figure 4.1, we can see that *maxSD* and *maxAvgSD* are faster than *dom* and *dom/deg* after the one-second threshold. When the algorithms caused the same number of failures, *maxSD* and *maxAvgSD* can solve more instances. *maxSD* and *maxAvgSD* solve around 80 percent of instances but the *dom* and *dom/deg* solve only around 60 percent of instances in the time limit. When we separate the aim50, aim100 and aim200 results to Figure 4.2, Figure 4.3 and Figure 4.4, we can clearly see that *maxSD* and *maxAvgSD* are more efficient to solve the moderate difficult instances like *aim100* or *aim200*. And even inside *aim100* and *aim200* our algorithm is better at solving problems with a large amount of failures. This phenomenon can be explained for the simple reason that our algorithm is meant to choose accurately the value on which you should search but takes a lot of processing time to find this value. This problem has around $m = 5n$ constraints and $t = \frac{1}{10}n$ tuples, then we can deduce that the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^3)$ ($w = 2$ here), compared to $O(n)$ for *dom* and *dom/deg*. Hence, for smaller examples the simple algorithms are faster because finding the solution might only require a few backtracks; on the other hand, with heavy examples, backtracks can be large and this is where the accurate choices have an impact on the number of failures and so on the efficiency to solve problems.

We can deduce that in the case of an instance with a medium number of variables, a low arity, a binary domain, a small number of relations, small tuples and medium constraints, our algorithm might be efficient. We will make other deductions with other examples to narrow down the parameters that allow our algorithm to be efficient.

We also observe that when the number of variables and constraints becomes greater but the number of tuples, and arity, and domain stays small (similar range of values), our algorithm performs better. Although, having a large number of constraints and variables might slow down our algorithm.

Renault Megane Configuration Problem

We tested 50 instances and each instance has around 100 variables and the domain of variables ranges from 0 to more than 30. The arity is between 2 and 10.

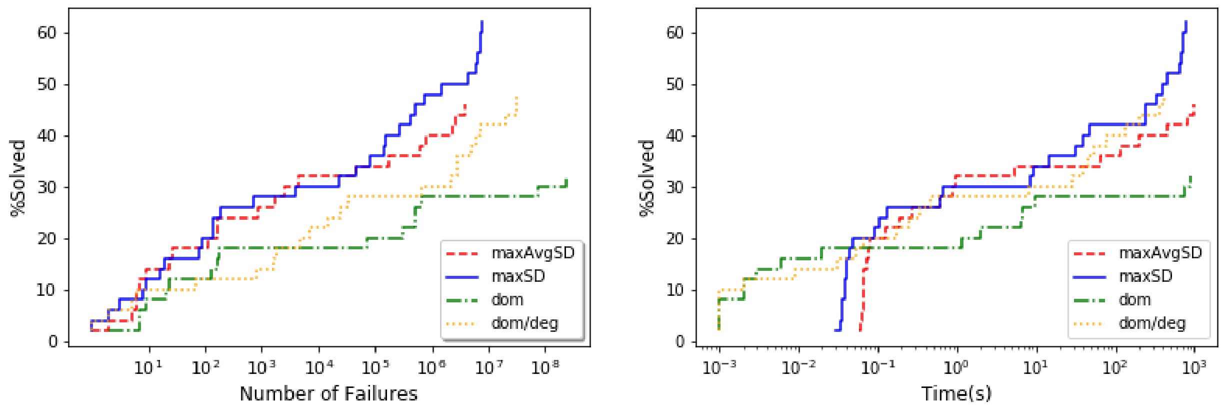

Figure 4.5 Percentage of modRenault instances solved w.r.t. time and number of failures

Here we observed that the *maxSD* can solve 60 percent of instances while the *dom* can only solve around 30 percent of instances. We can see $maxSD$ and $maxAvgSD$ are better than dom and dom/deg in the meanwhile $maxSD$ can solve more instances.

Something that might be interesting to notice in this example is that this time *maxAvgSD* and *maxSD* do not follow the same trends. In fact, *maxAvgSD* is slower than *maxSD* in this example. The differences we cannotice from this example and *aim100* are the domain and number of tuples. In this problem, the domain is not binary and it has large tables. In fact, this problem has around $m = n$ constraints and in the worst case $t = 20n$ tuples, then the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^4)$ $(w = \frac{1}{5}n)$. Hence we will have more processing to do in the *maxAvgSD* search.

When the instances have a medium number of variables, low arity, a large number of tuples and a medium number of constraints, we can guess that our algorithm *maxSD* works well and *maxAvgSD* is still efficient.

Kakuro puzzle problem

We tested 192 instances of this problem. The domain of each variable is small (around 9) and number of variables (up to 150) is large.

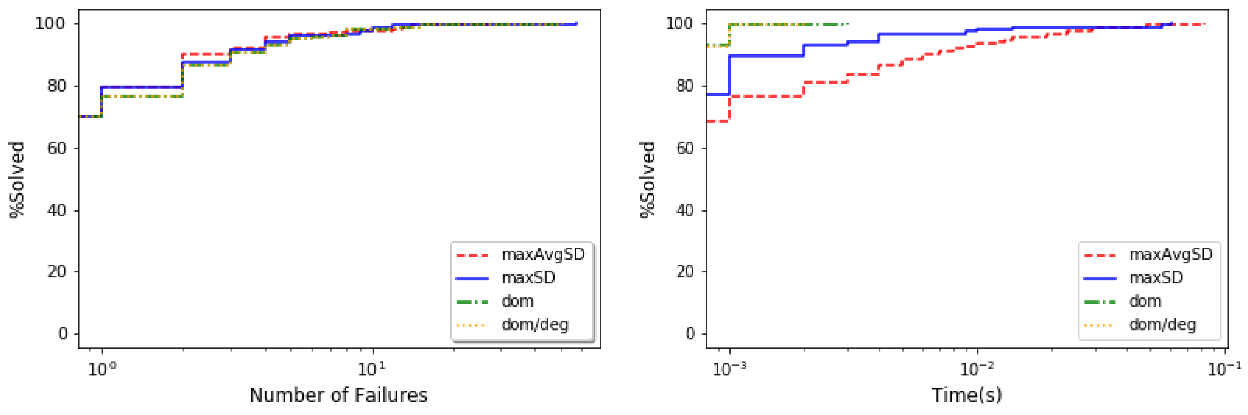

Figure 4.6 Percentage of $kakuro-medium$ instances solved w.r.t. time and number of failures

This result is really interesting because *maxAvgSD*, *maxSD*, and *dom*, and *dom/deg* have a similar performance. The *dom* and *dom/deg* are faster than our algorithms. This problem has around $m = n$ constraints and in the worst case $t = 100n$ tuples, then the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^4)$ ($w = \frac{1}{10}n$), compared to $O(n)$ for *dom* and *dom/deg*. The *maxAvgSD* is slower than *maxSD*, it is because the number of values affected the performance of *maxAvgSD*.

Traveling Salesman Problem

We used *TSP20* as an experiment for this part. There are 229 constraints and 61 variables and the arity is small (either 2 or 3) in each instance.

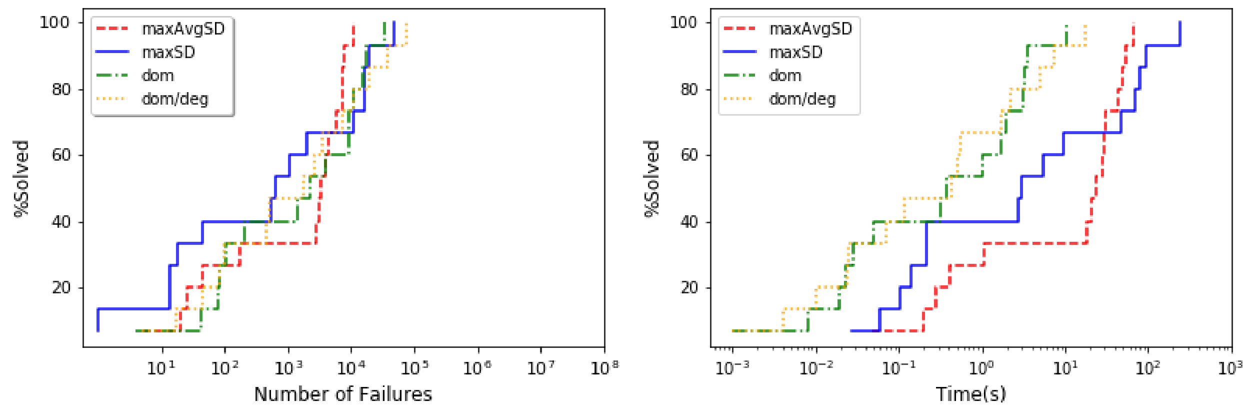

Figure 4.7 Percentage of TSP20 instances solved w.r.t. time and number of failures

Here we can observe that our algorithm lacks some optimization, the number of failures is similar between *maxAvgSD* and *dom* but *maxAvgSD* is slower. This problem has around $m = 4n$ constraints and $t = 200n$ tuples, then the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^4)$ ($w = 2n$), compared to $O(n)$ for *dom* and *dom/deg*. That might be the reason why *maxAvgSD* is much slower compared to *maxSD*. And this is also quite normal seeing how much computing we do compared to other simple searches. There is a sharp slope at the end of Figure 4.7 for *maxAvgSD*. Some variables occur in many constraints, so the *maxAvgSD* can have a better decision.

Dubois and A5

We tested 13 instances for Dubois and 50 instances for A5. Dubois has a quite small table and the arity is 3. A5 has a large table which has more than 10,000 tuples and the arity is 5 for all the instances.

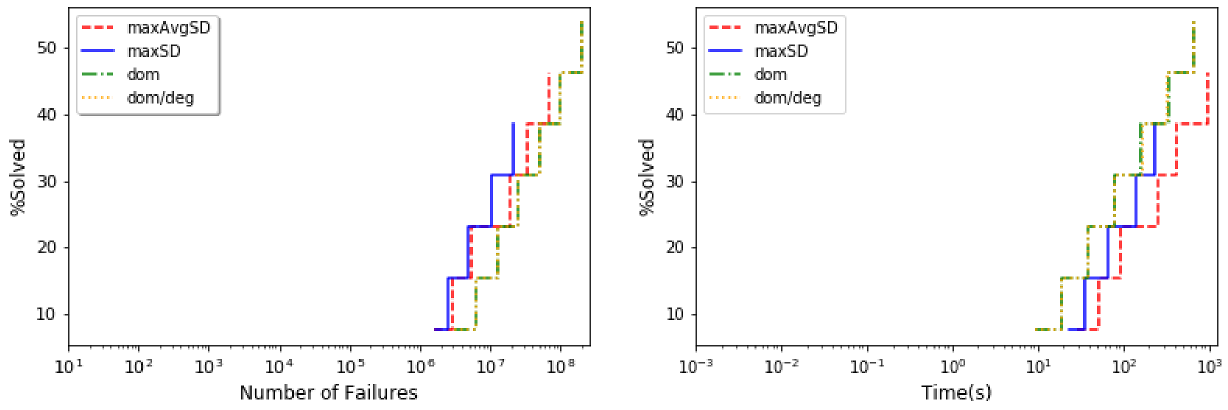

Figure 4.8 Percentage of Dubois instances solved w.r.t. time and number of failures

We can see in Fig. 4.8 that *dom* and *dom/deg* have the same number of failures and they can solve more than 50 percent of instances and $maxSD$ and $maxAvgSD$ cause fewer failures but solve fewer instances. The reason is that Dubois is hard to solve and our algorithms cannot solve it in the time limit. Because the number of failures is around 10^7, $maxSD$ caused much fewer failures compared to other algorithms.

Dubois maintains really small tables and it has many variables in each instance. The arity of the table is 3, so we only calculate solution densities for 3 variables over more than 60 variables. The domain of this problem is either 0 or 1, so the methods *dom* and *dom/deg* will struggle to decide between two variables. All these reasons might give our algorithm more probability to avoid more failures.

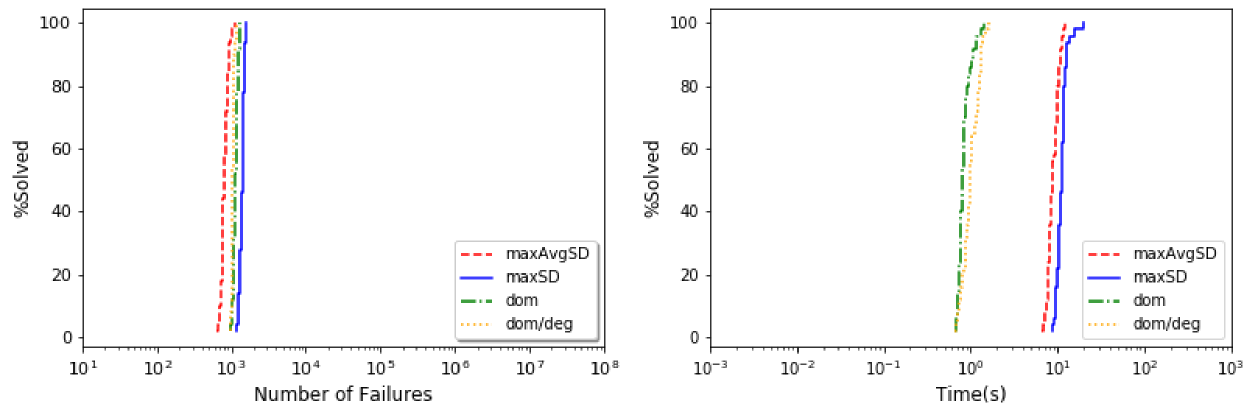

Figure 4.9 Percentage of a5 instances solved w.r.t. time and number of failures

For *a5*, we know $maxAvgSD$ considers the number of constraints and the number of constraints will make the $maxAvgSD$ and $maxSD$ have different performances. We can see in Fig. 4.9 that $maxAvgSD$ is better than $maxSD$. When we tracked one of the instances of this problem, we found that there are a lot of constraints related to variables. Maybe it will be a proof for our hypothesis that when there are a lot of constraints bound with each variable, $maxAvgSD$ can make a better decision compared to $maxSD$. Because of the time complexity of our algorithms, this allows us to see a difference in runtime between our heuristics and the others.

This problem has around $m = 5n$ constraints and $t = 1000n$, then the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^4)$ $(w = n)$, compared to O(n) for *dom* and *dom/deg*.

Nonogram

We tested 180 instances of this problem. The arity is large and there are a lot of variables in some instances and the number of constraints is not so large.

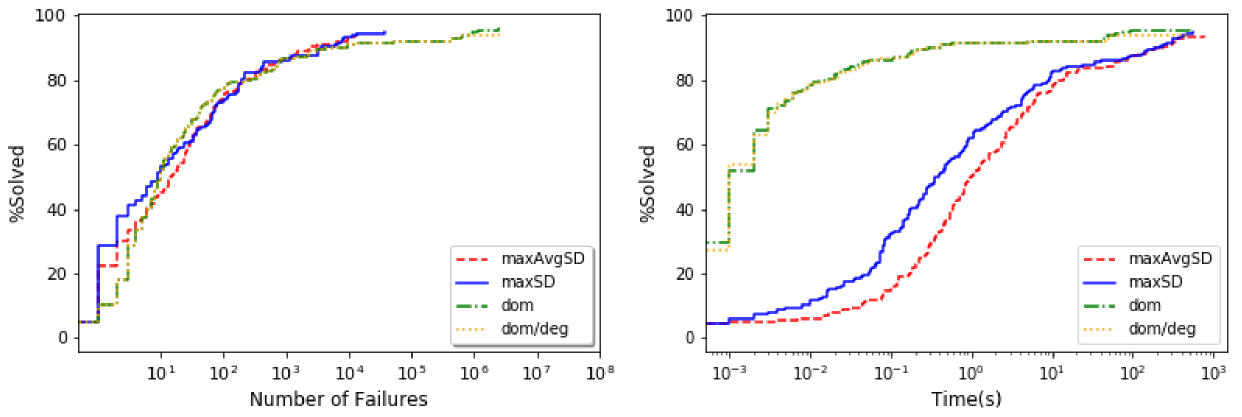

Figure 4.10 Percentage of nonogram instances solved w.r.t. time and number of failures

Figure 4.10 clearly shows that our algorithm is too time consuming for this problem. In a way, our algorithm cannot be as simple as *dom* algorithm, but we can see that for the same number of failures, our algorithm is still slower. This must come from the fact that we are doing a lot of processing to have the best choice of variable and value. In this problem, for example, having a simple heuristic to search might be useful because the choice will not impact the number of failures we will make in the end.

The arity of *Nonogram* is large and this problem has a lot of variables, so processing the solution density might have taken too long for this problem for our algorithms to be efficient. But we can also see that the computation time curves meet at the end, indicating that our branching heuristics become competitive for harder instances.

This problem has around $m = \frac{1}{5}n$ constraints and $t = 15n$, then the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^3)$ $(w = 2)$, compared to O(n) for *dom* and *dom/deg*.

Langford number problem

We tested the Langford and there are 20 instances. The domain is large (up to 40 values in the domain) and the number of constraints is huge (up to one thousand) and the arity is 3.

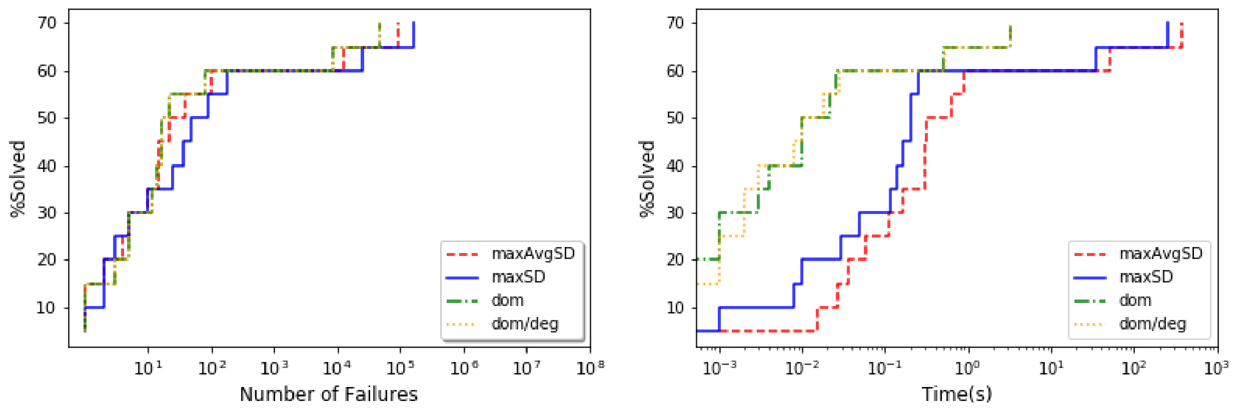

Figure 4.11 Percentage of langford2 instances solved w.r.t. time and number of failures

Here we know that our algorithm lacks some speed, the number of failures is similar among all four algorithms but our algorithms are slower. This problem has around $m = 20n$ constraints and $t = 30n$ tuples, then the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^4)$ $(w = n)$, compared to $O(n)$ for *dom* and *dom/deg*. The domain of variables is large in this problem, so the $maxAvgSD$ is slower than $maxSD$ even if $maxAvgSD$ generates fewer failures because its choices are better.

Crosswords problem

Here, we used $Crosswords\_lexVg$ and $Crosswords\_wordsVg$ to test all the algorithms. The $Crosswords\_lexVg$ and $Crosswords\_wordsVg$ both have small dictionaries and small tables and the $Crosswords\_lexVg$ used the dictionary defined by Stergiou [27]. Under Linux, the $Crosswords\_wordsVg$ used the dictionary in /usr/dict/words.

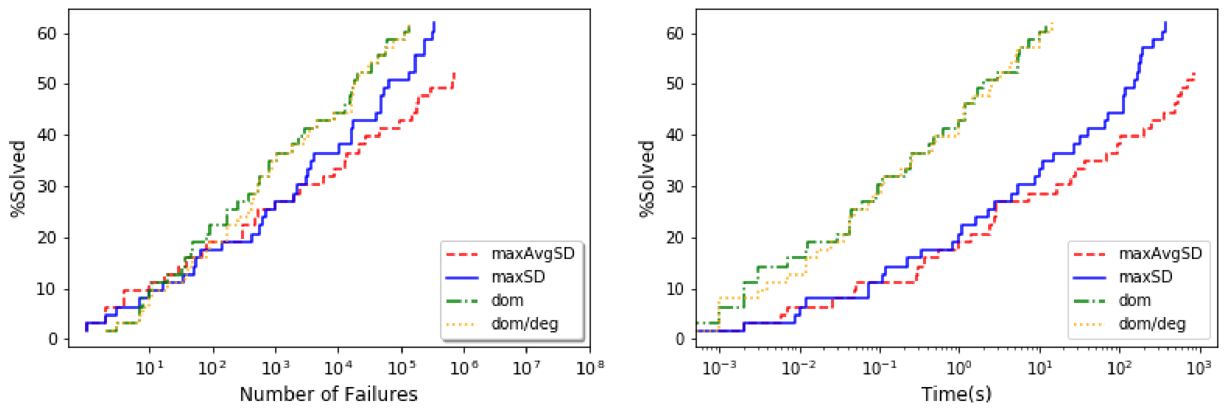

Figure 4.12 Percentage of $Crosswords\_lexVg$ instances solved w.r.t. time and number of failures

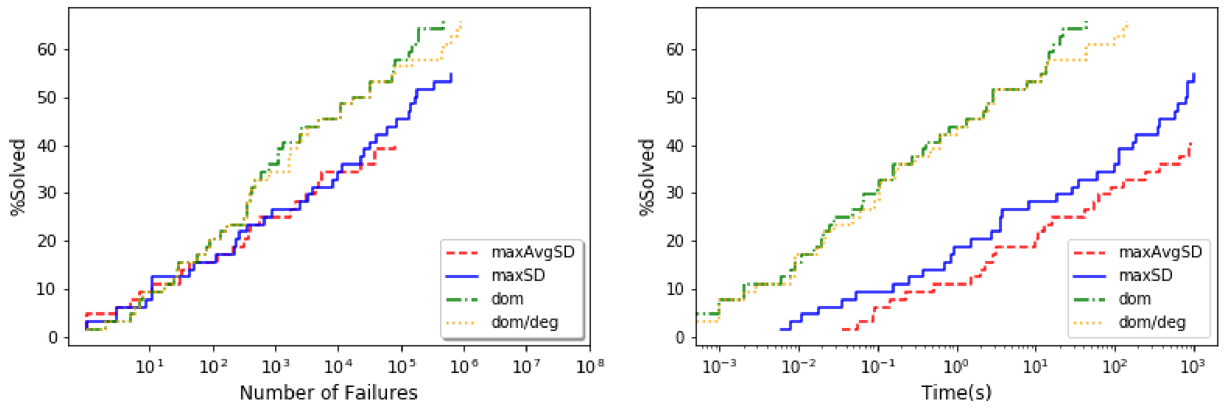

Figure 4.13 Percentage of $Crosswords\_wordsVg$ instances solved w.r.t. time and number of failures

We can easily see our methods created more failures and used more time. The $Crosswords\_lexVg$ and $Crosswords\_wordsVg$ have similar performance. The crossword problem instance has a large arity and a small number of constraints. We are not surprised that the performance is not good. We can then say that our algorithms are not worth using for problems with a large arity and a small number of constraints and that it might be better to use another heuristic in those cases.

This problem has around $m = \frac{1}{10}n$ constraints and $t = 10n$ tuples, then the complexity of *maxSD* in terms of the number of variables will be $O(n^3)$, and the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^4)$ ($w = \frac{1}{10}n$), compared to O(n) for *dom* and *dom*/*deg*.

Binary decision diagram

bddsmall are some instances of problems where the domain is binary. But the number of constraints and the number of variables is high.

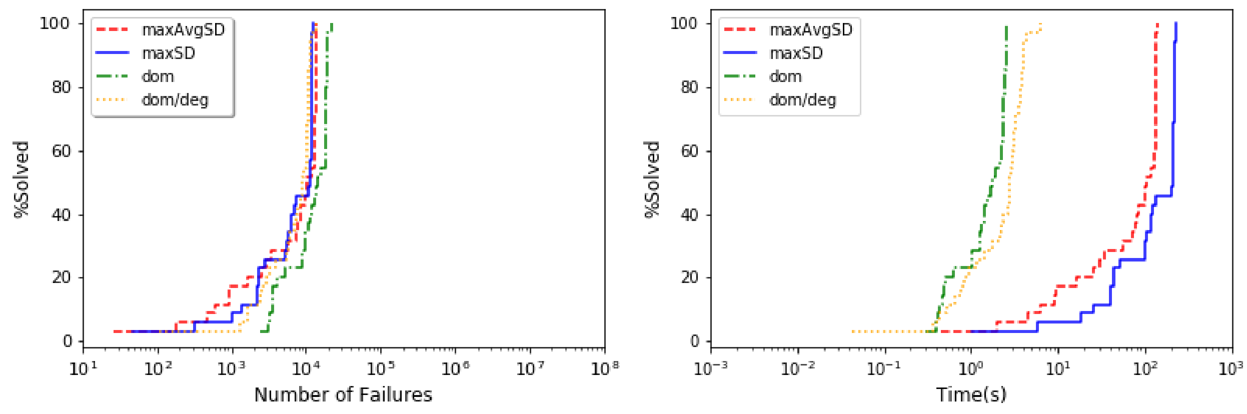

Figure 4.14 Percentage of bddsmall instances solved w.r.t. time and number of failures

The number of values does not affect the performance much, because the complexity of *maxAvgSD* in terms of the number of variables will be $O(n^3)$ ($w = 2$, $t = 3000n$), compared to $O(n)$ for *dom* and *dom*/*deg* and $O(n^3)$ for *maxSD*. This problem has around $m = 6n$ constraints, maxAvgSD might be able to make a better decision compared to *maxSD*. This problem has a large arity once again, so that might be the reason why our algorithm is not efficient.

Easy generation of hard (satisfiable) instances

All instances have 10 variables with a domain size of 10 and 15 table constraints of arity 5.

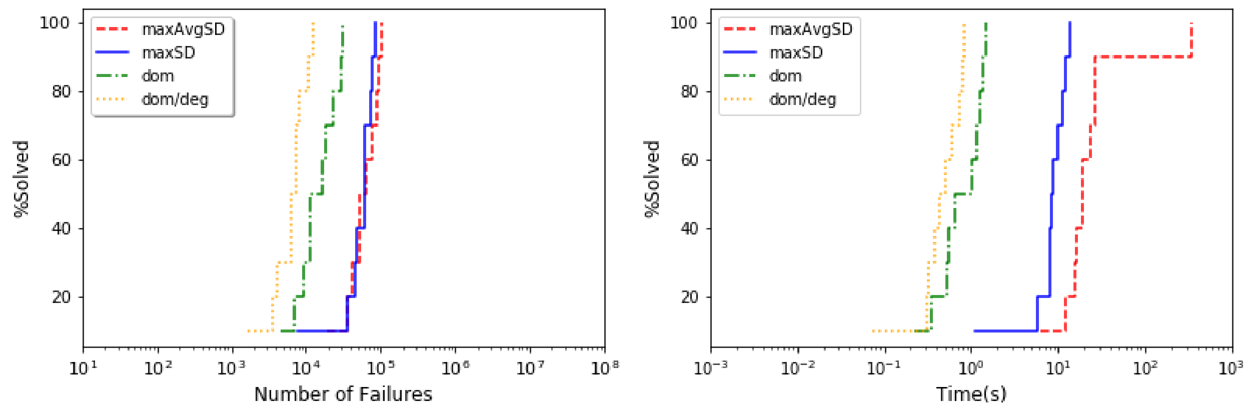

Figure 4.15 Percentage of $k5\_n10\_d10\_m15\_p08$ instances solved w.r.t. time and number of failures

Our algorithm is not fast for this kind of problem and we also generate more failures. The *dom/deg* can solve all the 10 instances within one second, but our most efficient algorithm *maxSD* needs 10 seconds. The *maxSD* and *maxAvgSD* have a similar number of failures but *maxAvgSD* is slower than *maxSD*. The complexity of *maxAvgSD* in terms of number of variables will be $O(n^4)$ ($w = n$, $t = 1000n$), compared to $O(n^3)$ for *maxSD*.

4.3 Conclusion with regards to the results

First, the implementation of counting-based search in Oscar is complex. It costs more time than the simple and efficient heuristic searches already implemented (such as dom) when the number of failures is similar. The update of the *mapcount* costs a lot of time. We maintained a *mapcount* to store the ID of variables and constraints and the support count. If the instance includes a big table, the update will be slower than the small table instance. In Subsection 4.2.3, some instances of the binary decision diagram include a large table with high arity. The update of *mapcount* used a long time, in the meanwhile, 40 percent of instances were solved immediately using *dom* and *deg/dom*. The results show us how much computation we did to maintain the *mapcount*.

Second, because we iterated all the values in the domain of variables in Algorithm 6 when the domain has a large number of values inside, *maxAvgSD* is slower than *maxSD*.

Third, we need to know the parameters in the instances are the number of variables and domain of variables and the number of tuples and the number of constraints and the arity of the relation. Each parameter affects the results of the instances. From Section 5.2, we can conclude our methods are efficient with many constraints when the arity is small. The domain of variables should be small too. If there are a lot of constraints involving a given variable, $maxAvgSD$ might give a better decision compared to $maxSD$.

Fourth, we clear the *mapcount* and save the solution density of unbound variables to improve the efficiency of the code. In this case, if the instance has a lot of constraints, the propagation will update almost all of the *mapcount* of variables and the improvement of code will be limited.

CHAPTER 5 CONCLUSION

In this thesis, we focused on counting-based search for compact table. We implemented CBS for compact table in Oscar and CBS performs well on several instances. In the next section, we will conclude the contributions of this thesis and then introduce the limitations of our implementation. Future research will be addressed at the end of this chapter.

5.1 Summary of Works

In Chapter 3, we introduced the compact table and the reversible sparse bit set and calculated the support count, which is the cornerstone of our implementation of CBS. We got the information from the compact table and considered the remaining values in the *mapcount*. In Chapter 4, the relationship between classes related to CBS is presented in a UML diagram. This UML of part of Oscar is useful for researchers who want to extend CBS in Oscar. We struggled a lot to add a unique ID for the variable and constraint. At the same time, another thing that bothered us was how to store and update the support count. We tried a lot of things to solve these two problems. At the same time, we understood that we had to make a few changes and use existing methods due to the complexity of Oscar. We improved the efficiency of the variable search by clearing the *mapcount* on time and saving the variable and solution density of the previous branching. To improve the efficiency of CBS, we also presented the variable search based on the arithmetic average and geometric average.

5.2 Limitations

In most of the cases, our algorithms were still slower than other heuristics even when sometimes we found a solution with fewer backtrack. This can be explained because our algorithm is not optimized enough. It takes some computation when we updated and stored the support count as well as when we searched on it. The heuristic search based on the domain of variables is easy to compute and it suits most of the constraints.

5.3 Future Research

As we mentioned in the limitations, we want to find a way to accelerate CBS and it will make CBS faster than the other heuristic search when our algorithms find a solution with fewer backtrack. Second, we put our algorithms in the cp/package as a bridge between constraint and branch. We would like to change CBS for the compact table to make it more elegant. Third, the update of *mapcount* used a lot of time even once we improved the efficiency of variable search. We would like to try to save the information about the compact table in the compact table itself while the information will not be erased in the execution, then CBS will be fast because we don't need to update the *mapcount* and we could find a solution on the fly.

REFERENCES

[1] OscaR Team. (2012) OscaR: Scala in OR. [Online]. Available: https://bitbucket.org/oscarlib/oscar

[2] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J. C. Régin, and P. Schaus, "Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets," presented at Principles and Practice of Constraint Programming, Toulouse, France, 05-09 Sep 2016, pp. 207–223. [Online]. Available: https://www.researchgate.net/publication/301878220_Compact-Table_Efficiently_Filtering_Table_Constraints_with_Reversible_Sparse_Bit-Sets

[3] G. Pesant and A. Zanarini, "Counting-based search: Branching heuristics for constraint satisfaction problems," *Journal of Artificial Intelligence Research*, vol. 43, pp. 173–210, Jan 2012. [Online]. Available: https://arxiv.org/abs/1401.4601

[4] A. Zanarini and G. Pesant, "More robust counting-based search heuristics with alldifferent constraints," presented at Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Bologna, Italy, 14-18 June 2010, pp. 354–368. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-13520-0_38

[5] G. Pesant and C. G. Quimper, "Counting solutions of knapsack constraints," presented at Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Paris, France, 20-23 May 2008, pp. 202–217. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-68155-7_17

[6] A. Lopez Ortiz, C. G. Quimper, J. Tromp, and P. Van Beek, "A fast and simple algorithm for bounds consistency of the alldifferent constraint," presented at 18th international joint conference on Artificial intelligence, Acapulco, Mexico, 09-15 Aug 2003, pp. 245–250. [Online]. Available: https://dl.acm.org/citation.cfm?id=1630695

[7] Gecode Team. (2017) Generic constraint development environment. [Online]. Available: https://www.gecode.org

[8] S. Gagnon and G. Pesant, "Accelerating counting-based search," presented at Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Delft, The Netherlands, 26-29 June 2018, pp. 245–253. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-93031-2_17

[9] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," presented at 6th international joint conference on Artificial intelligence, Tokyo, Japan, 20-23 Aug 1979, pp. 356–364. [Online]. Available: https://dl.acm.org/citation.cfm?id=1624861.1624942

[10] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," presented at 16th European Conference on Artificial Intelligence, Valencia, Spain, 20-27 Aug 2004, pp. 146–150. [Online]. Available: https://dl.acm.org/citation.cfm?id=3000033

[11] R. Philippe, "Impact-based search strategies for constraint programming," presented at Principles and Practice of Constraint Programming, Toronto, Canada, 27 Sep - 01 Oct 2004, pp. 557–571. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-30201-8_41

[12] L. Michel and P. Van Hentenryck, "Activity-based search for black-box contraint-programming solvers," presented at Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, Nantes, France, 28 May - 1 Jun 2012, pp. 228–243. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-29828-8_15

[13] B. M. Smith and P. Sturdy, "Value ordering for finding all solutions," presented at Principles and Practice of Constraint Programming, Perugia, Italy, 12-16 Sep 2011, pp. 606–620. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-23786-7_46

[14] D. Frost and R. Dechter, "Look-ahead value ordering for constraint satisfaction problems," presented at 14th international joint conference on Artificial intelligence, Montreal, Canada, 20-25, Aug 1995, pp. 572–578. [Online]. Available: https://dl.acm.org/citation.cfm?id=1625930

[15] C. Lecoutre and J. Vion, "Enforcing arc consistency using bitwise operations," *Constraint Programming Letters*, vol. 2, pp. 21–35, Jan 2008. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00868075/document

[16] C. Lecoutre and R. Szymanek, "Generalized arc consistency for positive table constraints," presented at Principles and Practice of Constraint Programming, Nantes, France, 25-29 Sep 2006, pp. 284–298. [Online]. Available: https://link.springer.com/chapter/10.1007/11889205_22

[17] P. Briggs and L. Torczon, "An efficient representation for sparse sets," *ACM Letters on Programming Languages and Systems*, vol. 2, pp. 59–69, Mar 1993. [Online]. Available: https://dl.acm.org/citation.cfm?id=176484

[18] V. le Clement de Saint Marcq, P. Schaus, C. Solnon, and C. Lecoutre, "Sparse-sets for domain implementation," presented at 19th International Conference on Principles and Practice of Constraint Programming, Uppsala, Sweden, 16-20 Sep 2013, pp. 1–10. [Online]. Available: https://www.info.ucl.ac.be/~pschaus/assets/publi/trics13-sparsets.pdf

[19] H. Verhaeghe, C. Lecoutre, and P. Schaus, "Extending compact-table to negative and short tables," presented at 31th AAAI Conference on Artificial Intelligence, San Francisco, USA, 4-9 Feb 2017, pp. 3961–3957. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/viewFile/14359/141220

[20] J. B. Mairy, Y. Deville, and C. Lecoutre, "The smart table constraint," presented at Integration of AI and OR Techniques in Constraint Programming, Barcelona, Spain, 18-22 May 2015, pp. 271–287. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-18008-3_19

[21] V. Hélène, C. Lecoutre, Y. Deville, and P. Schaus, "Extending compact-table to basic smart tables," presented at Principles and Practice of Constraint Programming, Melbourne, Australia, 28 Aug - 1 Sep 2017, pp. 297–307. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-66158-2_19

[22] L. Ingmar and C. Schulte, "Making compact-table compact," presented at Principles and Practice of Constraint Programming, Lille, France, 27-31 Aug 2018, pp. 210–218. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-98334-9_14

[23] S. Gay, R. Hartert, C. Lecoutre, and S. Pierre, "Conflict ordering search for scheduling problems," presented at Principles and Practice of Constraint Programming, Cork, Ireland, 31 Aug - 4 Sep 2015, pp. 140–148. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-98334-9_14

[24] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman, "Generating satisfiable problem instances," presented at 17th National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, Austin, USA, Jul 30-Aug 3 2000, pp. 256–261. [Online]. Available: https://dl.acm.org/citation.cfm?id=721445

[25] CSPLib Team. (2013) CSPLib: A problem library for constraints. [Online]. Available: http://www.csplib.org

[26] X. Ke, B. Frédéric, H. Fred, and L. Christophe, "Random constraint satisfaction: Easy generation of hard (satisfiable) instances," *Artificial Intelligence*, vol. 171, pp. 514–534, Jun 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370207000653

[27] N. Samaras and K. Stergiou, "Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results," *Journal Of Artificial Intelligence Research*, vol. 24, pp. 641–684, Nov 2005. [Online]. Available: https://arxiv.org/abs/1109.5714